

A programação em código de máquina não é usual, mas, para melhor resolução de determinados problemas, é vantajoso compreender como se pode utilizar essa linguagem. Ajudar o leitor nessa tarefa é o objectivo desta obra que, numa primeira parte, explica as instruções de Z80 Assembler e, numa segunda parte, trata da aplicação prática dessas instruções. O leitor encontrará aqui tabelas, programas e indicações para a manipulação do código de máquina, tendo todos os programas sido testados pelo próprio autor.



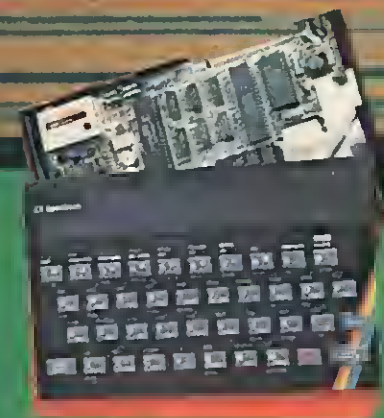
EDITORIAL PRESENÇA

Z80 Assembler Para o ZX SPECTRUM

154

JOÃO PAULO FRAGOSO

# Z80 ASSEMBLER PARA O ZX SPECTRUM introdução ao código de máquina



TEMPOS  
LIVRES

## CULTURA E TEMPOS LIVRES

1. ABC do Xadrez, *Petar Trifunovitch e Sava Vukovich*
4. ABC do Bridge, *Pierre Jais e H. Lahana*
5. Guia Prático de Fotografia, *W. D. Emanuel*
6. ABC do Judo, *E. J. Harrison*
7. Como Fazer Cinema, *Paul Petzold*
8. Bridge Moderno *Pierre Jais e H. Lahana*
9. Fotografia — Técnicas e Truques, *Edwin Smith*
10. ABC dos Estilos — da Arquitectura ao Mobiliário, *A. Aussel*
11. Fotografia — Técnicas e Truques, *Edwin Smith*
12. A Pesca Submarina, *António Ribera*
13. Teoria dos Finais de Partida, *Yuri Averbach*
14. Aprenda Rádio, *B. Fighiera*
15. Guia do Cão, *Louise Laliberté-Robert e Jean-Pierre Robert*
16. ABC do Aquário, *Anthony Evans*
17. Iniciação à Electricidade e Electrónica, *Fernand Huré*
18. Os Transístores, *Fernand Huré*
19. Karaté, I, *Albrecht Pflüger*
20. Iniciação ao Radiocomando dos Modelos Reduzidos, *C. Péronce*
21. Construa o seu Receptor, *B. Fighiera*
22. Montagens Electrónicas, *B. Fighiera*
23. O Berbequim Eléctrico, *Villy Dreier*
24. Cactos, *J. Nilaus Jensen*
25. Iniciação à Alta Fidelidade, *Peter Turner*
26. O Aquário de Água Doce, *Paulo de Oliveira*
27. ABC do Ténis, *Fonseca Vaz*
28. Karaté, II, *Albrecht Pflüger*
29. ABC da Criação de Canários, *Curt Af Enehjelm*
30. Ginástica Feminina, *Sonja Helmer Jensen*
31. Cartomancia, *Rhea Koch*
32. Calculadoras Electrónicas de Bolso, *E. Dam Ravn*
33. O Pastor Alemão, *Gilles Legrand*
34. Xadrez — Teoria do Meio Jogo I, *Bondarevsky*
35. Manual do Super 8, I, *Myron A. Matzkin*
36. ABC da Criação de Periquitos, *Cyril H. Rogers*
37. O Livro dos Gatos, *Barbel Gerber e Horst Bliefeld*
38. Manual do Super 8, II, *Myron A. Matzkin*
39. ABC do Mergulho Desportivo, *Walter Mattes*
40. Circuitos Integrados/Aplicações Práticas, *F. Bergtold*
41. A Apicultura, *H. R. C. Riches*
42. ABC do Cultivo das Plantas, *H. G. Witham Fogg*
43. ABC da Criação de Pombos, *Kai R. Dahl*
44. Construção de Caixas Acústicas de Alta Fidelidade, *R. Brautl*
45. Raças de Canários, *Klaus Speicher*
46. Jogos de Cartas, *Graciano Dolma*
47. Cocker Spaniels, *H. S. Lloyd*
48. ABC da Caça, *Fabian Abril*
49. Aprenda Televisão, *Gordon J. King*
50. Iniciação à Pesca, *Juan Nadal*
51. Basquetebol, *Marius Norregard*
52. Cães de Caça, *Santiago Pons*
53. Aprenda Electrónica, *T. L. Squires e C. M. Deason*
54. A Avicultura, *Jim Worthington*
55. A Produção de Coelho, *F. Surdeau e R. Henaff*
56. ABC dos Computadores, *T. F. Fry*
57. Natação para Crianças, *John Idorn*
58. O Boxer, *Anni Mortensen*
59. Voleibol, *Ole Hansen e Per-Göran Persson*
60. Iniciação à Vela, *Donald Law*
61. ABC da Filatelia, *Jacqueline Caurat*
62. A Pesca à Beira-Mar, *J.-M. Bœlle e B. Doyen*
63. Enxerto de Árvores de Fruto, *Alejo Rigau*
64. A Cultura do Moranguieiro, *L. A. Grau*
65. Emissores-Receptores (Walkies-Talkies), *P. Duranton*
66. Iniciação à Fotoelectrónica, *Heinz Richter*
67. Doces e Conservas de Frutas, *Robin Howe*
68. A Criação de Hamsters, *C. F. Snow*
69. A Criação de Porcos, *Roy Genders*
70. Calendário do Horticultor, *Luis Alsina Grau*
71. Jogos Electrónicos, *F. Rayer*
72. Cultivo de Cogumelos e Trufas, *A. Rigau*
73. Aprenda Televisão a Cortes, *G. J. King*
74. Gravação em Fita Magnética, *Ian R. Sinclair*
75. Poda de Árvores e Arbustos, *R. Genders*
76. Como Treinar o Seu Cão, *E. Fitch Daglish*
77. Instrumentos de Medida e Verificação, *Heinrich Stöckle*
78. A Criação de Caracóis, *Mañas Josa*
79. Rádio — Fundamentos e Técnicas, *Gordon J. King*
80. Como Fazer Gelados, *Sylvie Thiébaut*
81. Iniciação à Jardinagem, *Noel Clarasó*
82. A Congelação dos Alimentos, *S. Lapointe*
83. Windsurf — Prancha à Vela, *E. Prade*
84. Raças de Cães, *O. Hasselfeldt*
85. Rummy e Canasta, *Claus D. Grupp*
86. A Encadernação, *Annie Persuy*
87. Aprenda Electricidade, *Heinz Richter*
88. Taxidermia — Embalsamamento de Pássaros e Mamíferos, *Harry Hjortaa*
89. Jogging — Correr para Manter a Forma, *Werner Sonntag*
90. ABC da Cozinha Chinesa, *S. Richmond*
91. Jogos T.V., *C. Tavernier*
92. Amplificadores de Som, *Richard Zierl*
93. O Livro do Poker, *Claus D. Grupp*
94. Aprenda a Desenhar, *Rose-Marie Prémont e Nicole Philippi*
95. O Mini-trampolim na Escola, *Sonja Helmer Jensen e Klaus Dano*
96. Jogos de Luzes e Efeitos Sonoros para Guitarras, *B. Fighiera*
97. O Cultivo do Tomate, *Louis N. Flawn*
98. Filhas Solares, *F. Juster*
99. Criação Doméstica de Coelho, *C. F. Snow*
100. Iniciação ao Futebol, *W. Mänle e H. Arnold*
101. Horóscopos Chineses, *G. Haddenbach*
102. Guia Prático de Marcenaria, *C. H. Hayward*
103. Andebol, *Fritz e Peter Hattig*
104. Dispositivos Anti-Roubo, *H. Schreiber*
105. Perus, Pintadas e Codornizes, *J. Sauze*
106. Crepes, Doces e Salgados, *F. Arzel*
107. Aperitivos e Entradas, *Myrene Tiano*
108. Ténis de Mesa, *Leslie Woollard*
109. Aprenda Surf, *Rick Abbot e Mike Baker*
110. Futebol — Técnica e Tática, *Kurt Laval*
111. A Vaca Leiteira, *C. T. Whittemore*
112. O Cubo Mágico, *Josef Trajber*
113. O Perdigueiro Português, *José M. Correia*
114. Pizzas e Massas à Italiana, *Marie A. Ränk*
115. O Cubo para quem já o Faz, *Josef Trajber*
116. A Pirâmide Mágica, a Torre e o Barril do Diabo, *M. Mrowka e W. J. Weber*
117. Gansos e Patos, *Marie Mourthe*
118. Iniciação ao Kung-Fu, *A. P. Harrington*
119. Electrónica e Fotografia, *Hanns-Peter Siebert*
120. O Livro da Fortuna, *Douglas Hill*
121. Construção de um Alimentador de Corrente, *Waldemar Baitinger*
122. Hóquei em Patins, *Francisco Velasco*
123. Técnicas de Tiro, *Anton Kovacic*
124. Aprenda a Tricotar, *Uta Mix*
125. ABC da Patinagem, *Christa-Maria e Richard Kerler*
126. A Pesca e os seus Segredos, *Armand Deschamps*
127. O Osciloscópio, *R. Rateau*
128. Guia Prático da Banda do Cidadão, *J. M. Normand*
129. Sumos e Batidos, *Manfred Donderski*
130. Introdução à Programação de Microcomputadores, *Peter C. Sanderson*
131. Aprenda Croché, *Uta Mix*
132. ABC do Microprocessador, *P. Mélusson*
133. Guia Prático de Basic, *Roger Hunt*
134. Introdução à Electrónica Digital, *Ian Sinclair*
135. ABC do Vídeo, *David K. Matthewson*

136. Fotografia em Movimento, *Don Morley*
137. Guia de Cobol, *Ray Welland*
138. Fotografia a Pequena Distância, *Sidney F. Ray*
139. Guia Moderno da Canaricultura, *Manuel Gonçalves*
140. Minieletrónica para Amadores, *Heinz Richter*
141. ABC da Programação de Computadores, *John Shelley*
142. Tarot — O Futuro Pelas Cartas, *Edwin J. Nigg*
143. ABC da Equitação, *Dorothy Johnson*
144. Como Programar o seu ZX 81, *Patrick Gueulle*
145. 100 Avarias TV e a Maneira Prática de as Detectar, *P. Duranton*
146. ABC da Horticultura, *Louis Giordano*
147. Basic Para Microcomputadores, *A. P. Stephenson*
148. Como Programar o Seu ZX Spectrum, *Tim Hartnell e Dilwyn Jones*
149. Iniciação aos Motores Diesel, *David S. Maclean*
150. 60 jogos para o ZX Spectrum, *David Harwod*
151. As Linhas da Mão, *Rose Hubert*
152. Cozinha Italiana, *Rotrand Degner*
153. Manual do ZX Spectrum, *Simpson e Terrel*
154. Z80 Assembler para o ZX Spectrum — Iniciação ao Código de Máquina,  
*João Paulo Fragoso*
155. Aeróbica, *H. Schulz*
156. ABC do Atletismo, *Denis Watts*
157. 26 Programas Basic para Microcomputadores, *Derrick Daines*
158. Aprenda Pascal no seu Microcomputador, *Jeremy Ruston*
159. Guia Moderno da Suinicultura, *Colin Whittemore*
160. O Bar em Sua Casa — 888 Cocktails, *Aladar von Wesendonk*
161. Código de Máquina Para Principiantes, *James Walsh*
162. Código de Máquina Para Programadores Avançados, *Paul Holmes*
163. ABC da Fruticultura, *Henri Gosselin*
164. ABC da Canoagem, *Alan Hyde*
165. Guia de Fortran, *Philip Ridler*
166. Manual da Secretária, *Philippa Ramage*
167. ABC das Antenas, *Gordon J. King*
168. Programar Aventuras no Seu Computador, *Andrew Nelson*
169. Guia do Sinclair QL, *Boris Allan*
170. Novas Aventuras no Seu ZX Spectrum, *Peter Shaw e James Mortleman*
171. O Computador no Escritório, *John Shelley*
172. Sobrêmesas, *Fred Timber*
173. Rádio do Circuito Oscilante ao Receptor de Ondas Curtas, *Richard Zierl*
174. Xadrez: ABC das Aberturas, *V. N. Panore*
175. A Construção de Pequenos Transformadores, *M. Dourian e F. Juster*
176. Guia de Pascal, *David Watt*
177. O ZX Spectrum na Educação, *Tim Hasnel, C. Johnson e D. Valentine*
178. Iniciação ao Snooker, *John Pulman*
179. Circuitos com Diacs, Tiristores e Triacs, *Fritz Bergtold*

JOÃO PAULO FRAGOSO

# Z80 ASSEMBLER PARA O ZX SPECTRUM INICIAÇÃO AO CÓDIGO DE MÁQUINA

2.<sup>a</sup> edição

*Gostaria aqui de deixar o meu agradecimento a todos aqueles que me encorajaram a escrever este livro e para ele contribuíram com sugestões e conselhos, em particular o António Gouveia, o José Neves e o Luís Pinto.*

*Dedico este livro ao meu amigo Manuel José Franco, cujo apoio constante veio sempre na altura certa.*

© *Copyright by* João Paulo Fragoso e Editorial Presença  
Capa de Rogério Silva

Reservados todos os direitos  
para a língua portuguesa à  
EDITORIAL PRESENÇA, LDA.  
Rua Augusto Gil, 35-A – 1000 LISBOA

## INTRODUÇÃO

Penso que ninguém programa em código de máquina. Imagine meter na cabeça o significado de 256 números que, acrescidos de certas combinações de 2, 3 e até 4 desses códigos, correspondem a mais de 600 instruções que designam operações específicas simples!

No entanto, todas as aplicações do Spectrum são realizadas em código de máquina. Mal liga o computador, o vasto programa residente na ROM executa uma série de operações que culminam na mensagem de "copyright" e aguardam o seu comando.

Se acredita que este livro o vai ensinar a programar em código de máquina, desengane-se; não vou ser o primeiro a tentar uma tarefa que é absurda. Mas, se por outro lado, este volume o ajudar a compreender como utilizar código de máquina, o objectivo destas páginas terá sido atingido.

Encontrará aqui tabelas, programas e recomendações para a manipulação de código de máquina. Todos os programas que apresento foram cuidadosamente testados e funcionam da maneira descrita.

Este livro encontra-se dividido em duas partes: a primeira explicará as instruções de Z80 Assembler; a segunda tratará de aplicações de ordem prática.

Espero que esta obra lhe seja útil.

JOÃO PAULO FRAGOSO

## CAPÍTULO 1

### PROBLEMAS A ENFRENTAR

O microprocessador do ZX SPECTRUM dá pelo nome de Z80. Como todos os microprocessadores, o Z80 possui um conjunto de instruções e particularidades próprias. As instruções são na realidade ordens que o Z80 compreende como tarefas definidas a realizar.

Pense no Z80 como um escravo dedicado com um vocabulário limitado a 694 palavras. Como escravo dedicado que é, ele fará tudo o que lhe disser com o seu vocabulário, uma coisa de cada vez, seguindo as suas ordens à risca, pela sequência em que as deu. Infelizmente, o Z80 é um escravo estúpido pois não pode aprender mais instruções do que as que já sabe e se as suas ordens não fizerem sentido, ele seguiu-las-á, mesmo que isso signifique entrar em "crash". Mais ainda: se ele não fizer aquilo que você PENSOU e lhe disse para fazer, o erro será certamente seu, uma vez que a memória do Z80 é prodigiosa (teoricamente o Spectrum poderá receber uma média de aproximadamente 20000 instruções para execução sem interrupção).

Como mencionei na Introdução, o vocabulário do Z80 é constituído por números e combinações de números de 0 a 255. Uma

vez que tal vocabulário nada tem de humano, os construtores do Z80 — ZILOG — criaram uma série de mnemônicas que correspondem, na razão de um para um, ao vocabulário de números do Z80. Estas mnemônicas designam-se por LINGUAGEM ASSEMBLER do Z80. Será esta linguagem que examinaremos com algum detalhe ao longo destas páginas.

Se juntarmos uma série de instruções para o Z80 desempenhar uma tarefa específica obteremos aquilo a que se chama um PROGRAMA. (Na realidade um programa tem mais coisas além de instruções, mas lá chegaremos.)

Vejamos o seguinte programa em Assembler, que soma os valores contidos nos endereços 40000 e 40001 e coloca o resultado em 40002 (não se preocupe se ainda não o percebe; isto é um exemplo):

```
LD A, (40000)
LD B,A
LD A, (40001)
ADD A,B
LD (40002), A
RET
```

Se pensa que o Z80 percebe este programa está redondamente enganado. Este programa precisa de ser traduzido da linguagem Assembler para código de máquina. Esta função pode ser feita por dois processos básicos: 1) o programador enche-se de paciência e converte as mnemônicas nos números respectivos, fazendo depois POKES na memória do computador (ou utilizando um programa de BASIC para o fazer); 2) recorrer a um ASSEMBLADOR.

Seja qual for o caminho seguido, o que o Z80 vai ler é o seguinte:

```
00111010
01000000
10011100
01000111
00111010
01000001
10011100
10000000
00110010
01000010
10011100
11001001
```

Esta lista de números BINÁRIOS é equivalente ao programa em Assembler. Aqui, cada linha descreve um BYTE do programa. Se comparar as duas listagens do mesmo programa compreenderá por que motivo não se programa em CÓDIGO de MÁQUINA (também conhecido por CÓDIGO OBJECTO).

Note que cada BYTE é formado por oito BIT, que tomam o valor 0 ou 1. Se o programa em código de máquina fosse introduzido em memória, o Z80 seria capaz de o executar.

Mas não poderíamos encontrar outra forma de representação do código de máquina mais acessível? Sem dúvida! Uma vez que o Spectrum aceita números decimais directamente não poderia esta notação ser melhor? Por vezes sim, por vezes não. As vantagens da notação decimal são aparentes, pelo que vejamos os inconvenientes.

Um byte pode representar um número de 0 a 255. Dois byte podem representar um inteiro de 0 a 65535. Para mantermos os valores do programa acima, como representamos 40000 em dois byte? Onde fraccionar o número? Este número é representado por 64 no primeiro byte e 156 no segundo ( $64 + 256 * 156$  - veja a página 173 do manual do Spectrum). Mas... um momento; não deveria ser 156 no primeiro byte e 64 no segundo? Os autores do Z80, por qualquer motivo, decidiram o contrário, o que muitas vezes provoca situações de erro. Trataremos deste caso mais adiante.

Uma vez que o cálculo do fraccionamento de um número decimal é suficientemente complicado para permitir uma fonte de erros, os programadores recorreram a outra notação: base 16 ou hexadecimal. Neste sistema os algarismos de 0 a 9 correspondem aos da base 10. Os números 11, 12, 13, 14, e 15 são representados pelas letras A, B, C, D, E e F, respectivamente. Assim 40000 decimal equivale a 9C40 em hexadecimal. Não se preocupe com a conversão (de caminho leia o Apêndice E do Manual do Spectrum). Encontra nos Anexos deste volume uma tabela muito simples e prática para converter números decimais em hexadecimais e vice-versa. A grande vantagem desta notação reside em dois pequenos pormenores: 1) qualquer número até 255 pode ser representado por dois dígitos hexadecimais; 2) qualquer número superior a 255 e inferior a 65536 pode ser representado por quatro dígitos hexadecimais. O fraccionamento de #9C40 (40000) para podermos colocá-lo em dois byte é simples: #40 no primeiro byte (LSB) e #9C no segundo (MSB).

Depois disto tudo, talvez não fosse má ideia treinar um pouco com conversões entre números para as três bases que se usam com o Spectrum. Assim, ligue o seu e introduza o seguinte programa que converte qualquer número de qualquer base até 16 no seu correspondente em qualquer base até 16. Habitue-se a usar a tabela de conversão de números decimais/hexadecimais e confira o resultado a que chegou com o programa. Pratique também com números binários (não lhe fará mal).

#### PROGRAMA MUDA BASE

```
10 CLS: PRINT "ESTE PROGRAMA MUDA DE QUALQUER
BASE <= 16 PARA QUALQUER OUTRA BASE <= 16"
15 INPUT "QUAL É A BASE VELHA?";X$
20 LET E= 0
25 IF X$= "" THEN GO TO 10
30 GO SUB 185
```

```
35 LET B= N
40 IF N < 2 OR N > 16 THEN GO TO 10
45 INPUT "QUAL É O NÚMERO?";X$
50 IF X$= "" THEN GO TO 45
55 GO SUB 205
60 IF E= 1 THEN PRINT "ERRO": LET E= 0: GO TO 45
65 LET N1= N
70 PRINT X$;"NA BASE 10 É "; N1
75 IF N1 < 1000000 THEN GO TO 85
80 PRINT "O NÚMERO NA BASE 10 É > = 1000000, PO
DENDO POR ISTO OCORREREM ERROS"
85 INPUT "QUAL É A NOVA BASE?";X$
90 IF X$= "" THEN GO TO 85
95 GO SUB 185
100 LET B1= N: IF N < 2 OR N > 16 THEN GO TO 85
105 LET B$= ""
110 LET V= INT (N1 / B1)
115 LET R= N1 - V * B1
120 IF R > 9 THEN GO TO 140
125 LET B$= B$ + CHR$ (R + 48)
130 LET N1= V: IF V= 0 THEN GO TO 145
135 GO TO 110
140 LET R= R + 55: LET B$= B$ + CHR$ (R): LET N1=
V: IF V < > 0 THEN GO TO 110
145 PRINT "O NÚMERO NA BASE "; B1; " É ";
150 FOR J= LEN B$ TO 1 STEP - 1
155 PRINT B$(J);: NEXT J
160 PRINT
165 INPUT "MAIS NÚMEROS (S / N)?";X$
170 IF X= "S" THEN GO TO 10
175 IF X$= "N" THEN STOP
180 GO TO 165
185 LET N= 0
190 FOR J= 1 TO LEN (X$): LET D= CODE X$(J)
195 LET N= N * 10 + D - 48: NEXT J
200 RETURN
205 LET N= 0
```



```

210 FOR J= 1 TO LEN X$: LET D= CODE X$(J)
215 IF D > 47 AND D < 58 THEN LET D= D - 48: GO TO 230
220 IF D > 64 AND D < 71 THEN LET D= D - 55: GO TO 230
225 LET E= 1: RETURN
230 IF D > = B THEN LET E= 1: RETURN
235 LET N= N * B + D
240 NEXT J
245 RETURN

```

## CAPÍTULO 2

### LINGUAGENS DE ALTO E BAIXO NÍVEL

O programa Muda Base, no fim do capítulo anterior, está escrito em BASIC. Esta linguagem, que vem incluída na ROM do Spectrum, é conhecida como uma linguagem de ALTO NÍVEL. Outros exemplos são o PASCAL, o FORTH, o LISP, o COBOL.

O código de máquina e a Linguagem Assembler são, por seu lado, linguagens de BAIXO NÍVEL. A distinção é simples: quanto mais elevado for o nível da linguagem que consideramos, mais ela se aproxima de uma linguagem humana. Assim, a maioria dos comandos do BASIC são de fácil apreensão, pois o seu significado em Inglês aproxima-se da função que executam. Permitem, pois, descrever as tarefas segundo formas orientadas para a resolução de problemas, em vez de serem orientadas para o computador, como o Assembler. Geralmente, um comando de BASIC corresponde a uma série de instruções em Assembler.

Uma vez que a linguagem “falada” pelo Z80 é o código de máquina, esta foi apelidada de linguagem de nível zero. O Assembler, dado que tem uma correspondência de um para um com o código de máquina, mas já está um pouco mais perto do entendimento do comum dos mortais, é dita uma linguagem de nível um.

As linguagens de alto nível são transformadas em código de máquina por dois processos: 1) recorrendo a um intérprete, como no caso do Spectrum (o BASIC é interpretado em código de máquina); 2) recorrendo a um compilador (o programa em linguagem de alto nível é convertido em código de máquina antes de ser executado). Vejamos agora as vantagens e desvantagens de cada nível.

As linguagens de alto nível tem como vantagens:

- 1 — Os programas são mais fáceis de escrever;
- 2 — Os programas escrevem-se mais rapidamente;
- 3 — O programador não precisa de conhecer a configuração

interna do computador que usa;

4 — Teoricamente um programa escrito numa linguagem de alto nível é executável em qualquer computador que suporte essa linguagem (na prática, os construtores criam barreiras artificiais para evitar isto; no entanto, o programa será fácil de converter e modificar);

Mas se as linguagens de alto nível são tão vantajosas, para que serve preocuparmo-nos com linguagens de baixo nível? Exacto! Eis as desvantagens das linguagens de alto nível:

1 — Cada linguagem de alto nível tem um conjunto de regras complicadas para ser usada — a sua sintaxe. É como aprender uma língua estrangeira!

2 — Precisam de ser interpretadas ou compiladas;

3 — Cada linguagem de alto nível foi criada com um fim particular em vista: O FORTRAN é uma linguagem virada para o cálculo, sendo muito difícil fazer processamento de “strings” de caracteres ou criar gráficos e desenhos com ela.

4 — As linguagens de alto nível produzem programas ineficientes, pois o interpretador ou o compilador tem de prever todas as situações realizáveis. Isto traduz-se na dificuldade de otimizar o código produzido e no sub-aproveitamento da memória disponível.

5 — Certas características do computador não podem ser usadas a partir da linguagem de alto nível (como, por exemplo, gravar programas sem “header” ou alterar a velocidade de gravação).

Por seu lado, as linguagens de baixo nível tem as seguintes vantagens principais:

1 — Qualquer que seja a tarefa a programar, o tipo de dificuldade é semelhante.

2 — O programa é eficiente, tanto no aproveitamento da memória e do código, como na rapidez de execução.

3 — Todas as características do computador são acessíveis. Mas nem tudo são rosas. Eis as desvantagens:

1 — Fazer com que o código reflita o problema. As instruções de código fazem coisas como colocar um valor num registo, girar os bit de um byte para a esquerda, comparar um registo com outro, etc. e não ver se uma tecla foi premida, ou se a sua nave espacial foi destruída por um míssil.

2 — Exige um grau mais ou menos detalhado do funcionamento do computador, por exemplo, quantos registos tem o microprocessador, que instruções usa e quais são os seus efeitos precisos, etc.

3 — Não permite ser usada noutro computador com um microprocessador diferente (e muitas vezes nem num com um igual, devido ao design do “hardware”). Ex: um programa de Z80 não é compreendido por um 6502 e vice-versa.

4 — É mais moroso programar numa linguagem de baixo nível.

Se é assim, para que serve preocuparmo-nos com código de máquina? A resposta é relativa. Como frisei atrás, ninguém programa em código de máquina (é por isto que há ASSEMBLADORES que são o tema do próximo capítulo).

Usa-se a programação em Assembler nos seguintes casos:

1 — Programas curtos a médios.

2 — Programas que devam otimizar o consumo de memória.

3 — Aplicações de tempo real.

4 — Processamento limitado de dados.

5 — Aplicações de grande volume.

6 — Mais input/output do que cálculos.

Usam-se linguagens de alto nível para:

1 — Programas longos.

2 — Aplicações de baixo volume que exijam grandes programas.

3 — Programas que exijam vastas memórias.

4 — Mais cálculo do que input/output.

De um modo geral, use linguagem Assembler se está interessado em economia de memória e de tempo (mas prepare-se para perder mais tempo no desenvolvimento do "software"). Se está interessado em economizar tempo no desenvolvimento do programa e não na rapidez de execução e uso eficiente da memória então use linguagens de alto nível.

Se pensa que pintei um quadro muito negro, deverá crer que não foi exagerado. Dá trabalho usar linguagens de baixo nível! Mas também compensa!

## CAPÍTULO 3

### ASSEMBLADORES

Partindo do princípio que continua interessado em programar o seu Spectrum em Assembler, terá de arranjar um programa utilitário, que dá pelo nome apropriado de ASSEMBLADOR.

Um assembler permite que o utilizador faça um programa, consistindo em linhas de instruções em Assembler (chamado programa FONTE ou "source", em Inglês). Quando estiver satisfeito com a fonte produzida, o assembler transforma a fonte em instruções de código objecto, prontas a ser executadas pelo Z80. Mas os assembladores disponíveis no mercado fazem mais do que esta mera conversão. Alguns são programas de alta sofisticação, que permitem trabalhos de carácter profissional, ao passo que os outros estão mais virados para o utilizador que dá os seus primeiros passos na programação em Assembler. Consequentemente, estes últimos são menos versáteis, mas mais fáceis de utilizar.

Independentemente do Assembler que escolher (comentarei alguns mais adiante), encontrará em todos eles determinadas características comuns. A primeira é a formatação de cada linha de instruções.

Assim, cada linha de um Assembler tem QUATRO CAMPOS ("fields", no Inglês):

- 1 — NOMES ("labels")
- 2 — CÓDIGO DE OPERAÇÃO ou CAMPO DE MNEMÓNICA
- 3 — OPERANDO ou CAMPO DE ENDEREÇAMENTO
- 4 — CAMPO DE COMENTÁRIOS

Os campos dos nomes e dos comentários são opcionais. O campo do operando pode ter um endereço, um dado ou estar vazio. O campo de mnemónica nunca pode estar vazio: ou tem uma mnemónica, ou uma DIRECTIVA ao assembler (que também dá pelos nomes de pseudo-instrução, pseudo-operação ou pseudo-op). Os diferentes campos ocorrem numa linha na ordem especificada e são separados uns dos outros por DELIMITADORES. O delimitador mais comum é o espaço. A propósito: tenha cuidado com eles. Embora haja um padrão, cada assembler tem as suas manias. O padrão Z80 para delimitadores (ou separadores) é o seguinte:

- : — após um nome
- espaço — entre o código de operação e o endereço
- ; — entre operandos no campo de endereçamento
- ; — antes de um comentário

Vejam os então os diferentes campos em maior detalhe. O primeiro é o campo dos nomes. Pode não ter nada e o Assembler não protestará se nunca o usar. Mas se o fizer, só poderá ganhar. Se um nome estiver presente neste campo, o assembler associa esse nome ao endereço da memória para onde o primeiro byte de código objecto dessa instrução for colocado (se não percebeu, leia de novo, devagar). Dar um nome permite que não nos preocupemos para onde esse código vai ser colocado. Se quisermos referi-lo, bastará indicar o nome que lhe demos. O assembler substituirá o nome pelo seu valor quando criar o código de máquina correspondente à fonte. Vejam os então as vantagens dos nomes:

- 1 — Permitem achar rapidamente um local do programa (e de nos lembramos dele).
- 2 — Podemos mover um nome para modificar ou corrigir um programa, sem termos de mexer nas outras instruções, pois o assembler tratará disso.

3 — Adicionando uma constante, o assembler poderá colocar o programa noutro ponto da memória que nos pareça mais conveniente.

4 — Toma-se fácil incorporar o programa noutro.

5 — Não precisamos de andar a contar byte nem de calcular onde é que certas instruções vão parar.

6 — Identificamos claramente uma instrução que queremos usar como destino ou que queremos realçar.

Uma vez que os nomes são tão úteis, quais vamos usar? Em primeiro lugar, nem todos os assemblers permitem nomes do mesmo comprimento: no EDITOR /ASSEMBLER da PICTURES-QUE o nome tem um máximo de 5 caracteres; no GENS da HISOFT o máximo é seis. Em segundo lugar é inútil (para não dizer ridículo) usarmos nomes como JOÃO ou LISBOA; devemos escolher nomes que nos fomeçam uma indicação do objectivo deles — nomes mnemónicos, como SOMA, MNSGM (para mensagem), etc. Deve evitar nomes do tipo XPTR, N3249 ou similares, pois em nada ajudam a documentar o programa fonte (e a documentação É importante).

Para terminarmos este tópico, eis algumas 'regras' para o uso de nomes:

- 1 — Use nomes diferentes de mnemónicas Z80.
- 2 — Não use nomes mais compridos do que o permitido pelo assembler.
- 3 — Use apenas maiúsculas ou maiúsculas seguidas de números (ex: MNSGM1, MNSGM2, MNSGM3, LIMECR)
- 4 — Comece os nomes com letras; tais nomes são sempre aceites.
- 5 — Evite caracteres especiais (ex: £, &, #, ©) e nomes facilmente confundíveis, como XXXX e XXXXX.
- 6 — Se não tem a certeza de que um certo nome é aceitável não o use.

O campo de mnemónica ou de códigos de operação é o campo fulcral de uma linha de Assembler. É este campo que determina a formação do código de máquina que o assembler produz. Ora,

acontece que há instruções de 1, 2, 3 e até 4 byte de comprimento. Qualquer assembler distingue-as, recorrendo a uma tabela interna. Veremos adiante quais são as instruções.

Falámos atrás em PSEUDO-OPERAÇÕES. Estas são instruções que não são transformadas em código de máquina, antes representam DIRECTIVAS ou comandos, que o assembler interpreta para realizar certas funções. As directivas que poderá encontrar são variadas. De entre as mais comuns (e úteis), destacamos:

ORG (de origem)  
EQU (de "equate")  
DEF (de define)

A directiva ORG permite-nos localizar o programa ou subrotinas ou mesmo dados em qualquer ponto da memória. Deve ser colocada no início da rotina que queremos colocar num local específico da memória e tem um argumento, que é esse local de memória. EX: ORG 40000 indica ao assembler que as linhas de Assembler seguintes deverão ser compiladas para operarem a partir do endereço 40000. Use sempre esta directiva.

A directiva EQU associa o valor do seu argumento ao nome que a precede. Como tal, esta directiva é desprovida de significado se não for precedida de um nome, uma vez que o define. Veja a seguinte linha de Assembler:

INÍCIO EQU 40000

A partir do momento em que o assembler lê esta linha, ele reserva numa tabela especial (tabela dos símbolos) espaço para associar INÍCIO a 40000. Usa-se para definir parâmetros e constantes. Sempre que o assembler encontrar, no nosso exemplo, o nome INÍCIO, "saberá" que lhe corresponde o valor 40000. O melhor lugar para colocar este tipo de directiva é o início do programa.

A directiva DEF é, na realidade, um grupo de quatro directivas:

1 — DEFB 2 — DEFW 3 — DEFS 4 — DEFM

DEFB quer dizer DEFine Byte. Deve geralmente ser precedida de um nome e reserva um byte, cujo conteúdo será o seu argumento. Note que pode ser qualquer inteiro até 255, inclusive. Exemplo:

CONTAD DEFB # 20

DEFW significa DEFine palavra ("Word"). É igual a DEFB, mas reservando 2 byte, permitindo um número de 0 a 65535. Exemplo:

LINHAS DEFW #FE35

DEFS quer dizer DEFine espaço ("space"). Também conhecida como RESERVE, pois reserva X byte, sendo X o argumento. Exemplo:

UDG001 DEFS 256

Neste exemplo, o assembler reservaria 256 byte a partir do presente local, cujo início teria o nome UDG001.

DEFM significa DEFine Mensagem. Na realidade, esta directiva é seguida por aspas, delimitando uma string, cujo tamanho máximo varia conforme o assembler. Exemplo:

MNSGM6 DEFM "Esta é a última directiva deste tipo"

Há ainda um outro tipo de directivas: as PSEUDO-MNEMÓNICAS CONDICIONAIS. O seu uso não é recomendável, constituindo uma ótima fonte de erros, quando usadas.

O campo de endereçamento é usado para os argumentos e endereços. A maior parte dos assembladores permite uma grande liberdade neste campo, pois as entradas podem ser em decimal, binário, hexadecimal ou nomes pré-definidos ou a definir. No caso

das entradas numéricas, os assembladores exigem geralmente que se preceda ou siga o endereço ou dado com uma letra ou carácter identificador da base do número. Assim, os mais comuns são:

BINÁRIO — B ou %

HEXADECIMAL — H ou \$ ou #

DECIMAL — D ou nada

Para além dos nomes, podemos ter 'strings', expressões algébricas ou lógicas e "offsets" em relação ao Ponteiro de Localização. Recomenda-se que:

- 1 — Use sempre o sistema de numeração mais claro para dados. Evite misturar num programa números em decimal com binário ou hexadecimal.
- 2 — Não confunda dados com endereços.
- 3 — Não use 'offsets' em relação ao Ponteiro de Localização.
- 4 — Evite expressões complicadas e opções do assemblador que não percebe ou são obscuras.

Qualquer assemblador permite-lhe pôr comentários no programa fonte. Estes comentários não têm efeito no código gerado, apenas visam explicar e documentar (e muitas vezes perceber) o programa. Assim, não tenha medo de tecer os seus comentários, mas:

- 1 — Não comente as instruções; diga o que o programa faz. Evite coisas como "SOMA 1 AO ACUMULADOR" ou "COMPARA O REGISTO B COM O ACUMULADOR" mas ponha coisas como "LEITURA DO TECLADO" ou como "FINALIZAÇÃO DO JOGO".
- 2 — Os comentários devem ser breves e claros. Não faça narrativas.
- 3 — Comente os pontos fulcrais do programa, bem como as partes que não são óbvias. Não comente mudanças normais de ponteiros ou contadores.

4 — Evite abreviações nos comentários.

5 — Deixe comentários do tipo anotação quando trabalha num programa. Ajudá-lo-ão a retomar o trabalho quando voltar, após interrupção.

6 — Seja claro! Um programa bem comentado facilita o trabalho.

Os assembladores para o Spectrum que se vendem por aí são assembladores de DUAS PASSAGENS, isto é, passam duas vezes pelo programa fonte. Na primeira recolhem e identificam todos os símbolos usados, verificando a sintaxe e a correcção da escrita das mnemónicas. A segunda passagem é que faz a produção de código de máquina.

O DEVPAK, nomeadamente a versão 3, é um conjunto de dois programas da HISOFT. O primeiro chama-se GENS (GENS2 ou GENS3, conforme a versão) e é um assemblador. O segundo é o MONS (MONS2, MONS3) e é um monitor. O conjunto é o melhor e o mais versátil do mercado actual, mas exige uma leitura dos manuais (que não são pequenos) antes de se obter um à-vontade razoável. Ambos os programas são relocatáveis (podem ser colocados e funcionar em qualquer lugar da RAM). É um conjunto de nível profissional.

O EDITOR /ASSEMBLER é o assemblador da PICTURES-QUE. É bastante bom, mas não é relocatável. É um bom assemblador para principiantes. Há também um monitor da mesma empresa, que é comercializado separadamente.

O ASPECT é o assemblador da ARTIC. Muito bom para principiantes, é o mais simples dos três para começar, embora seja o de nível inferior.

Há mais, mas estes são os que conheço. A decisão cabe-lhe a si.

## CAPÍTULO 4

### O Z80 / REGISTOS E 'FLAGS'

Como foi dito atrás, a programação em Assembler exige algum conhecimento do "hardware" envolvido. Veremos agora o Z80 nesses detalhes necessários.

Tal como precisamos de uma memória para nos recordarmos daquilo que nos é dito de imediato, o Z80 tem uma espécie de memória a curto-prazo 'similar'. Esta memória é o conjunto dos seus REGISTOS.

O Z80 dispõe de SETE registos directamente manipuláveis, identificados abaixo:

A (ou Acumulador)  
B C  
D E  
H L

O acumulador é o registo privilegiado do Z80, uma espécie de mão direita. É mais rápido, faz mais coisas. Todas as operações aritméticas e de lógica Booleana usam o Acumulador como operando e destino final.

Os registos B, C, D, E, H e L são os registos secundários. Embora a sua manipulação seja idêntica, há diferenças importantes entre eles. Podem ser utilizados aos pares (BC, DE e HL) para operações e manipulação de dados de 16 - bit. Neste caso particular o par HL é privilegiado (se está a pensar por que motivo se chama HL e não FG a razão é simples: para ajudar os programadores - na língua inglesa - convencionou-se assim para os lembrar da ORDEM do byte - H (high) é o superior e L (low) é o inferior). Há instruções que usam os pares BC e DE como ponteiros de dados de 16 - bit.

Temos ainda outros registos de uso limitado: os registos de indexamento IX e IY. São registos de 16 - bit. Nota importante: salvo casos especiais (isto é, se já domina Assembler), não use o IY, senão arrisca-se a ter um "crash" no programa. O Spectrum precisa dele para outros fins.

Finalmente há registos de uso limitadíssimo: R e I. O registo R (refrescamento) pode ser lido para obter um número aleatório entre 0 e 255. Fora disso é preferível deixá-lo em paz. O registo I (vector de Interrupção) pode ser usado (e é - o bastante em protecção de "software" comercial), mas a nota que leu referida ao IY aplica-se ao registo I com a mesma justeza, pois valores entre 64 e 127 provocam interferências no écran.

Seguem-se as "flags" (sinalizadores é a melhor tradução que se pode arranjar, mas flag é preferível). As flags ocupam o equivalente a outro registo — apropriadamente F — que emparelha com o Acumulador. Cada Flag tem um bit, assumindo por isso os valores 0 ou 1. O Z80 tem 6 flags (os construtores acharam que eram suficientes). Eis as suas designações:

BIT	SÍMBOLO	NOME
0	C	CARRY
1	N	SUBTRACT (subtracção)
2	P / O	PARITY / OVERFLOW
3	-	----
4	A / C	AUXILIARY CARRY
5	-	----
6	Z	ZERO
7	S	SIGN (sinal)

A CARRY (transporte) transporta o bit mais significativo de qualquer operação aritmética. É usada nas instruções SHIFT. Toma o valor 0 (reset) após operações Booleanas.

A SUBTRACT foi feita para facilitar a vida ao Z80. Toma o valor 0 com todas as instruções de adição e o valor 1 com todas as instruções de subtração.

A PARITY / OVERFLOW (paridade / excesso) é uma flag de vários usos, dependendo da operação a ser executada: se são operações aritméticas é uma flag de excesso; para input, rotação e operações Booleanas é uma flag de paridade, tomando o valor 1 se a paridade for par e o valor 0 se a paridade for ímpar. Tem o valor 1 durante a transferência de blocos e operações de busca, voltando a zero quando o contador de byte chega a zero. Também é afectada por interrupções (IFF2), aquando da execução das instruções LD A,I e LD A,R.

A flag ZERO toma o valor 1 com quaisquer operações aritméticas ou Booleanas que gerem um resultado nulo e toma o valor 0 quando essas operações geram um resultado não nulo.

A flag SIGN (sinal) toma o valor do bit mais significativo do resultado da execução de qualquer operação aritmética ou Booleana.

Finalmente a AUXILIARY CARRY ("carry" auxiliar) faz o transporte do bit 3 para o bit 4 que resulte de operações aritméticas.

Note que todas as flags mantêm o último valor até que a execução de uma nova instrução as modifique e desde que essa instrução tenha efeito sobre elas.

Seguem-se mais dois "registos" especiais: o CONTADOR DE PROGRAMA, abreviado PC (de "Program Counter") e o PONTEIRO da "PILHA", abreviado SP (de "STACK Pointer"). Têm ambos 16 bit.

O PC é o indicador que o Z80 possui para saber qual a instrução que está a executar. Assume o valor do endereço e é incrementado automaticamente no número de byte correcto ou modificado para seguir as instruções de salto (JUMP).

O SP permite-lhe criar o Stack em qualquer local da memória de RAM. É o "machine stack" que pode ver na pág. 165 do seu manual. Embora não haja limite para ele (a não ser a memória disponível), raramente necessitará mais de 256 byte. É usado para ter acesso a subrotinas e processar interrupções. NÃO USE O SP

PARA PASSAR PARÂMETROS PARA AS SUBROTINAS. O Stack começa no seu endereço mais elevado, crescendo para baixo. Instruções PUSH decrementam o conteúdo do SP e as instruções POP incrementam o seu conteúdo.

Se o que foi dito até aqui lhe parece um pouco complicado, não se preocupe. Pode aproveitar para reler agora (e sempre que quiser ou necessitar). De qualquer modo ficou já com uma ideia do Z80, nos seus detalhes estruturais. Depois de alguns projectos de programação, estas informações ficarão cimentadas na sua memória.

O Z80 tem vários modos de endereçamento de memória, a saber:

- 1 — IMPLÍCITO
- 3 — TRANSFERÊNCIA DE BLOCOS IMPLÍCITA COM DECREMENTAÇÃO OU INCREMENTAÇÃO AUTOMÁTICA
- 4 — IMPLÍCITO AO STACK
- 5 — INDEXADO
- 6 — DIRECTO
- 7 — RELATIVO AO PROGRAMA
- 8 — PAGINADO DE BASE
- 9 — INDIRECTO AOS REGISTOS
- 10 — IMEDIATO

Não nos deteremos com estes modos. Basta saber que se referem a grupos de instruções, cujo significado poderá ser obtido em obras mais avançadas do que esta.

Para finalizar este capítulo, vamos ver um pouco de registos alternativos.

Suponha que tem todos os registos ocupados com dados intermediários de um cálculo e, antes de prosseguir, é necessário usá-los para algumas operações. Suponha ainda que não tem memória disponível para o fazer ou que o tempo de execução que pretende do programa é o mínimo. Felizmente para si, o Z80 possui REGISTOS ALTERNATIVOS, que se deferenciam dos "principais" seguindo-os de um apóstrofo. São eles:



F' A'  
 B' C'  
 D' E'  
 H' L'

Por meio da instrução EXX poderá passar de um conjunto para outro, mas não misturá-los ou usá-los em simultâneo.

Nos capítulos seguintes veremos exemplos de instruções e o seu significado.

## CAPÍTULO 5

### LOAD

Neste capítulo veremos um grupo de instruções que permitem colocar valores nos registos ou na memória. As instruções serão descritas do seguinte modo: o seu tipo geral, a função que realizam, as flags que afectam. Não nos deteremos em cada uma das 694 instruções em particular, nem referiremos os ciclos de relógio (o tempo de execução) que cada uma leva, salvo nos casos em que isso possa ser vantajoso para já.

Recomenda-se a consulta do Glossário de Termos e Abreviações, no fim do livro, antes de prosseguir.

TIPO: LD reg, data EXEMPLO: LD A, # FF

FUNÇÃO: Esta instrução é lida como “guarda no registo reg o valor data”. Depois da execução da instrução exemplificada o Acumulador conterá o valor 255 decimal (FF hexadecimal). Ocupa 2 byte.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD par, data 16 EXEMPLO: LD DE, # 9C40

**FUNÇÃO:** Esta instrução é lida como “guarda no par de registos “par” o valor data 16”. Depois da execução da instrução exemplificada o registo D terá o valor #9C e o registo E terá o valor # 40. Ocupa 3 byte. Note que a instrução LD SP, data 16 lhe permite inicializar o seu Stack em qualquer ponto da memória RAM.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** LD A,(end)                      **EXEMPLO:** LD A,(# 9C40 )

**FUNÇÃO:** Esta instrução é lida como “guarda no Acumulador o valor contido no byte endereçado por end”. Depois da execução da instrução exemplificada e supondo que o conteúdo do byte com o endereço # 9C40 é # 2F, o conteúdo do Acumulador é perdido e passará a ser # 2F (até voltar a ser alterado). O byte # 9C40 manterá o seu valor de # 2F. Ocupa 3 byte. Note que este é um dos casos de privilégio do Acumulador.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** LD par, (end)                      **EXEMPLO:** LD BC,( # 9C40)  
LD indreg, (end)                      LD IX,( # 7400)

**FUNÇÃO:** Esta instrução é lida como “guarda no par de registos (no registo de indexamento) o valor contido no byte endereçado por end e o valor contido no byte end + 1”. Suponha que o byte endereçado por # 9C40 contém # A0 e o byte # 9C41 contém # 00. Após a execução da instrução exemplificada em primeiro lugar, o registo B conterá # A0 e o registo C conterá # 00. Note que estas instruções usam 4 byte e levam 20 ciclos a serem executadas, ao passo que a instrução privilegiada LD HL,(end) ocupa apenas 3 byte e leva 16 ciclos a ser executada. É assim preferível usar esta última sempre que possível.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** LD (end), A                      **EXEMPLO:** LD (# 9C40 ),A.

**FUNÇÃO:** Esta instrução é lida como “guarda no Acumulador o valor contido no byte endereçado por end”. Suponha que # 9C40 contém # 36. Após a execução da instrução exemplificada o Acumulador terá o valor # 36, perdendo-se o seu anterior conteúdo. Note de novo o privilégio do Acumulador e a simetria do vocabulário do Z80.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** LD (end), par                      **EXEMPLO:** LD (# A000),DE  
LD (end), indreg                      LD (# C500),IX

**FUNÇÃO:** Esta instrução é lida como “guarda no par de registos (registo de indexamento) o valor de 16-bit contido nos byte endereçados por end e end + 1”. Suponha que # A000 contém # 20 e A001 contém # 34. Após a execução da instrução exemplificada, o registo D terá o valor # 34 e o registo E o valor # 20. Note que o par privilegiado HL, neste tipo de instrução, usa apenas 3 byte e 16 ciclos para a execução, ao passo que qualquer outro par ou registo de indexamento necessita de 4 byte e 20 ciclos de tempo real de execução.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** LD reg, reg                      **EXEMPLO:** LD A,B

**FUNÇÃO:** Esta instrução é lida como “guarda no registo reg o conteúdo do registo reg”. Suponha que B contém # 76. Após a execução da instrução exemplificada o Acumulador conterá igualmente # 76.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** LD reg,(HL)                      **EXEMPLO:** LD C,(HL)  
LD reg, (indreg + d)                      LD E,(IX + 5)

**FUNÇÃO:** Esta instrução é lida como “guarda no registo reg o valor contido no byte endereçado por HL. Suponha que HL contém o endereço # 9000 e esse byte contém # 17. Após a execução

da instrução exemplificada o registo C terá o valor # 17. Note que no caso dos registos de indexamento o valor d de deslocamento terá de estar sempre presente (nem que seja zero). Note ainda que usando HL usa apenas 1 byte e 7 ciclos, ao passo que com os registos de indexamento usa 3 byte e 19 ciclos.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD (HL),reg                      EXEMPLO: LD (HL),B  
LD (indreg + d), reg                      LD (IX + 4), D

FUNÇÃO: Esta instrução é lida como “guarda no byte endereçado por HL o valor contido no registo reg”. Note que é o grupo simétrico do tipo que vimos acima. Suponha que IX contém o valor #A000 e que o byte endereçado por # A004 contém # 03. Após execução da instrução exemplificada, o registo D conterá # 03.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD (HL), dado                      EXEMPLO: LD (HL),# 6B  
LD (indreg + d),dado                      LD (IX + 255),# 3F

FUNÇÃO: Esta instrução é lida como “guarda no byte endereçado por HL (registo de indexamento mais deslocado) o valor dado”. Suponha que HL contém : FFFF. Após a execução da instrução exemplificada o byte endereçado por HL,# FFFF, conterá # 6B. Note que o uso da instrução que recorre a HL necessita de 2 byte e 10 ciclos, enquanto que as que usam os registos de indexamento precisam de 4 byte e 19 ciclos.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD A,(BC) e LD A,(DE)

FUNÇÃO: Esta instrução é lida como “guarda no Acumulador o valor contido no byte endereçado por BC (ou DE)”.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD (BC),A e LD (DE),A

FUNÇÃO: Esta instrução é lida como “guarda no byte endereçado pelo par BC (ou DE) o valor contido no Acumulador”. Note que estas instruções são simétricas das anteriores.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD SP,HL e LD SP,indreg

FUNÇÃO: Esta instrução é lida como “coloca o valor de 16-bit contido no par HL (ou num dos registos de indexamento) no Ponteiro do Stack”.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: LD I,A e LD R,A

FUNÇÃO: Esta instrução é lida como “guarda no registo Interrupção (ou Refrescamento) o valor contido no Acumulador”. Não se esqueça de que mexer no registo I poderá resultar em perda de qualidade da imagem.

FLAGS: não são afectadas por este tipo de instrução.

TIPO: LD A,I e LD A,R

FUNÇÃO: Esta instrução é lida como “guarda no Acumulador o valor contido no registo I (ou R)”. Note que, apesar de serem as instruções simétricas das anteriores, estas AFECTAM as flags.

FLAGS: C- NA Z- X S- X P/0-1 AC-0 N-0

Vimos, de modo abreviado, toda as instruções que permitem colocar um valor num local de memória ou num registo. Repara-

mos que, com excepção de duas instruções, este grupo não afecta as flags. O capítulo seguinte apresentará as instruções que executam operações aritméticas.

## CAPÍTULO 6

### OPERAÇÕES ARITMÉTICAS

O Z80 efectua operações aritméticas simples — adição e subtracção — com inteiros de 8 e 16 bit. Operações mais complexas como a divisão e a multiplicação podem ser realizadas com instruções de rotação e mudança, que veremos adiante, ou com loops de adições e subtracções. Gostaria de salientar que o papel das flags não é, de modo algum, desprezável para este tipo de instruções. Repare somente no seguinte caso: suponha que quer somar o conteúdo de dois registos e colocar o total num terceiro. É o caso de operação soma em 8 bit. Se um registo contiver # 80 e o outro # C4, o resultado obtido é superior a # FF, o que implica a impossibilidade de o armazenar num registo de um byte: é uma situação de OVERFLOW. Sem a indicação da flag não saberíamos que o número realmente colocado no terceiro registo estava errado. Por isto (e por outros motivos que se tornarão evidentes), muito cuidado com operações aritméticas!

#### OPERAÇÕES ARITMÉTICAS DE 8 BIT

TIPO: ADD A,dado

EXEMPLO: ADD A,#20



**FUNÇÃO:** Esta instrução é lida como “subtraia o conteúdo do registo reg ao conteúdo do Acumulador e coloque o resultado neste”. Similar a ADD A,reg para a subtracção.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: SUB (HL) e SUB (indreg + d)

**FUNÇÃO:** Esta instrução é lida como “subtraia o valor contido no byte endereçado por HL (ou indreg mais d) ao conteúdo do Acumulador, colocando o resultado neste”. Similar a ADD (HL) e a ADD (indreg + d), mas para a subtracção. Note que os comentários referentes ao número de byte e aos ciclos mantêm-se válidos.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: SBC A,data

EXEMPLO: SBC A,# 80

**FUNÇÃO:** Esta instrução quer dizer “subtraia o valor data e o estado da Carry ao conteúdo do Acumulador, colocando neste o resultado”. Suponha que o Acumulador tem o valor # 7F e a Carry está set. Após a execução da instrução exemplificada o Acumulador conterá # 00. Similar a ADC data, mas para a subtracção.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: SBC A,reg

EXEMPLO: SBC A,H

**FUNÇÃO:** Esta instrução significa “subtraia o conteúdo do registo reg e o estado da Carry ao conteúdo do Acumulador, guardando o resultado neste”. Similar a ADC reg, mas para a subtracção.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: SBC A,(HL) e SBC A,(indreg + d)

**FUNÇÃO:** Esta instrução quer dizer “subtraia o conteúdo do byte endereçado por HL (ou por indreg + d) e o estado da Carry ao conteúdo do Acumulador, guardando neste o resultado”. Similar a ADC A,(HL), e a ADC A,(indreg + d), mas para a subtracção. Note que os comentários referentes aos byte e ciclos de relógio necessários também aqui têm relevância.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: CP data

EXEMPLO: CP #20

**FUNÇÃO:** Esta instrução é lida como “compara o valor data com o Acumulador, alterando as flags de acordo com a comparação”. Se cada vez que quiséssemos comparar dois números tivéssemos de usar instruções de subtracção, perderíamos o conteúdo do Acumulador. Tal não acontece com a instrução CP, que efectua a subtracção mas despreza o resultado, alterando apenas as flags. As comparações usam o Acumulador, sendo Z= 1 para teste de igualdade, Z= 0 para a desigualdade, C= 1 para conteúdo do Acumulador menor que data C= 0 para conteúdo do Acumulador maior ou igual a data. Se o Acumulador contivesse # 20, a instrução exemplificada faria Z= 1 e C= 0.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: CP reg

EXEMPLO: CP L

**FUNÇÃO:** Esta instrução é lida como “compara o conteúdo do registo reg com o conteúdo do Acumulador, alterando as flags de acordo com o resultado da comparação”. Similar a CP data para registos.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: CP (HL) e CP (indreg + d)

**FUNÇÃO:** Esta instrução quer dizer “compara o conteúdo do byte-endereçado por HL (ou por indreg + d) com o conteúdo do

Acumulador, alterando as flags de acordo com a comparação''. Similar a CP data e a CP reg, mas para referência indirecta. Note que CP (HL) necessita de 1 byte e 7 ciclos, ao passo que CP (indreg + d) precisa de 3 byte e 19 ciclos.

FLAGS: C- X Z- X S- X P/O- X AC- X N- 1

TIPO: INC reg

EXEMPLO: INC C

FUNÇÃO: Esta instrução significa "incrementa em 1 o conteúdo do registo reg". Se C contiver # F5, após a execução da instrução exemplificada, passará a ter # F6.

FLAGS: C- NA Z- X S- X P/O- 0 AC- X N- 0

TIPO: DEC reg

EXEMPLO: DEC A

FUNÇÃO: Esta instrução significa "decrementa em 1 o conteúdo do registo reg". Se A contém # 81, após a execução da instrução exemplificada, passará a ter # 80.

FLAGS: C- NA Z- X S- X P/O- 0 AC- X N- 1

## OPERAÇÕES ARITMÉTICAS DE 16 BIT

TIPO: ADD HL,par

EXEMPLO: ADD HL,HL

FUNÇÃO: Esta instrução significa "soma o conteúdo do par de registos especificado ao conteúdo do par HL". É similar a ADD A,reg só que para 16-bit. Se HL contiver # 1000, depois da execução da instrução exemplificada HL conterà # 2000 (equivalente a multiplicar por dois).

FLAGS: C- X Z- NA S- NA P/O- NA AC- ? N- 0

TIPO: ADD IX,pp

EXEMPLO: ADD IX,BC

FUNÇÃO: Esta instrução quer dizer "soma o conteúdo de pp (onde pp= BC, DE, IX ou SP) ao conteúdo de IX". Similar a ADD HL,par.

FLAGS: C- X Z- NA S- NA P/O- NA AC- ? N- 0

TIPO: ADD IY,rr

EXEMPLO: ADD IY,SP

FUNÇÃO: Esta instrução é lida como "soma o conteúdo de rr (onde rr= BC, DE, IY ou SP) ao conteúdo de IY". Similar a ADD HL, par.

FLAGS: C- X Z- NA S- NA P/O- NA AC- ? N- 0

TIPO: ADC HL,par

EXEMPLO: ADC HL,DE

FUNÇÃO: Esta instrução significa "soma o conteúdo do par de registo par e o estado da Carry ao par HL". É o equivalente, para 16-bit, de ADC A,reg.

FLAGS: C- X Z- X S- X P/O- X AC- ? N- 0

TIPO: SBC HL,par

EXEMPLO: SBC HL,DE

FUNÇÃO: Esta instrução é lida como "subtrai o conteúdo do par de registos par e o estado da Carry ao conteúdo do par HL". Esta instrução é o equivalente de 16-bit à instrução SBC A,reg.

FLAGS: C- X Z- X S- X P/O- X AC- ? N- 1

TIPO: INC par

EXEMPLO: INC HL

INC indreg

INC IX

FUNÇÃO: Esta instrução significa "incrementa em 1 o conteúdo do par de registos par (ou registo de indexamento indreg)". Note que são os correspondentes de 16 bit a INC reg. No entanto, NÃO AFECTAM AS FLAGS. Note ainda que o uso de INC indreg implica mais um byte e mais 4 ciclos do que INC par.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: INC (HL) e INC (indreg + d)

FUNÇÃO: Esta instrução quer dizer “incremente em 1 o conteúdo do byte endereçado por HL (ou por indreg + d)”. Note que a instrução com HL necessita de apenas 1 ciclo e 1 byte, ao passo que a que usa os registos de indexamento necessitam de 23 ciclos e 3 byte.

FLAGS: C- NA Z- X S- X P/O- 0 AC- X N- 0

TIPO: DEC par  
DEC indreg

EXEMPLO: DEC DE  
DEC IY

FUNÇÃO: Esta instrução quer dizer “decremente em 1 o conteúdo do par de registos par (ou registo de indexamento indreg)”. É a instrução correspondente, em 16-bit, a DEC reg. Os comentários feitos acerca da instrução INC par são também válidos aqui.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: DEC (HL) e DEC (indreg + d)

FUNÇÃO: Esta instrução quer dizer “decremente em 1 o conteúdo do byte endereçado por HL (ou por indreg + d)”. Note que a instrução com HL necessita de apenas 1 ciclo e 1 byte, ao passo que a que usa os registos de indexamento necessita de 23 ciclos e 3 byte.

FLAGS: C- NA Z- X S- X P/O- 0 AC- X N- 1

Como se viu, o conjunto de instruções para lidar com números de 16-bit é bem mais limitado do que o conjunto que lida com números de 8-bit.

É boa altura para introduzir o conceito de número negativo. Pudemos constatar que o Z80 trabalha com inteiros positivos. No

entanto, quando programamos, podemos convencionar que metade da gama de números disponíveis é positiva e a outra metade (a segunda, para ser preciso) é negativa. A tabela de conversão de números decimais em hexadecimais leva esta convenção em devida conta. Na prática, a programação em Assembler dispensa-nos a preocupação de qual o sinal do algarismo em questão. Todos os algarismos negativos, de acordo com a convenção, tem o bit de ordem mais elevada em 1.

O capítulo seguinte considerará um grupo mais pequeno de instruções, mas muito usado: as instruções Booleanas ou instruções lógicas.



## CAPÍTULO 7

### INSTRUÇÕES LÓGICAS

As operações lógicas que o Z80 realiza estão compreendidas na álgebra Booleana. Vá ver o seu Manual do Spectrum na página 86, do capítulo 13, onde encontra explicadas as operações de BASIC AND, OR e NOT. O Z80 tem qualquer coisa semelhante ao nível de bit, mas permitindo mais opções lógicas.

#### Grupo AND

Suponha que tem dois byte, A e B. O byte A tem o valor 10100000B e o byte B=11100011B. A instrução AND compara, para cada posição de bit, os bit respectivos de cada valor comparado, tendo resultado 1 nesse bit se ambos os valores tiverem esse bit = 1 e zero em todos os outros casos. Assim, no nosso exemplo o resultado será 10100000B.

```

      B= 11100011
      A= 10100000
      -----
      10100000
  
```

Note que, prosseguindo no exemplo e considerando que A é o Acumulador e B o registo B, o resultado viria em A e as flags reflectiriam o resultado, nomeadamente:

AC= 1, C= 0 e N= 0  
 P / O= 1 (há paridade — número par de bit com valor 1)  
 S= 1 (o bit mais significativo tem valor 1)  
 Z= 0 (o resultado é diferente de zero)

TIPO: AND data

EXEMPLO: AND #0D

**FUNÇÃO:** Esta instrução significa “AND o conteúdo do próximo byte de memória com o Acumulador (colocando neste o resultado)”.

FLAGS: C- 0    Z- X    S- X    P/O- P    AC- 1    N- 0  
 Note que P significa paridade

TIPO: AND reg

EXEMPLO: AND B

**FUNÇÃO:** Esta instrução quer dizer “AND o conteúdo do registo reg com o do Acumulador, guardando neste o resultado”.

FLAGS: C- 0    Z- X    S- X    P/O- P    AC- 1    N- 0

TIPO: AND (HL)

AND (indreg + d)

EXEMPLO: AND (IY + 34)

**FUNÇÃO:** Esta instrução significa “AND o conteúdo do byte endereçado por HL (ou indreg + d) com o conteúdo do Acumulador, colocando neste o resultado”.

FLAGS: C- 0    Z- X    S- X    P/O- P    AC- 1    N- 0

#### Grupo OR

Suponha de novo que temos os dois valores A= 00111010B e B= 01111100B. A instrução compara os valores de modo semelhante a AND, mas dando o valor 1 se ambos os bit forem 1 ou se

um for 1 e o outro 0 ou o valor zero no caso de ambos os bit terem o valor 0. Assim, o exemplo dará 0111110B.

$$\begin{array}{r} A = 00111010 \\ B = 01111100 \\ \hline 01111110 \end{array}$$

TIPO: OR data

EXEMPLO: OR # 7C

FUNÇÃO: Esta instrução significa "OR o valor do byte de memória que se segue à mnemónica com o valor do Acumulador, colocando neste o resultado".

FLAGS: C-0 Z-X S-X P/O-P AC-1 N-0

TIPO: OR reg

EXEMPLO: OR B

FUNÇÃO: Esta instrução quer dizer "OR o conteúdo do registo reg com o conteúdo do Acumulador, guardando neste o resultado".

FLAGS: C-0 Z-X S-X P/O-P AC-1 N-0

TIPO: OR (HL)

OR (indreg + d)

EXEMPLO: OR (IY + 80)

FUNÇÃO: Esta instrução significa "OR o conteúdo do byte endereçado por HL (ou por indreg + d) com o conteúdo do Acumulador, colocando o resultado neste".

FLAGS: C-0 Z-X S-X P/O-P AC-1 N-0

Grupo XOR

Considere os byte A = 00111010B e B = 01111100B. A operação lógica XOR (de eXclusive OR) é similar às operações ante-

riores, mas o valor do bit é 1 só quando um dos bit é 1 e o outro é zero. Todos os outros casos dão valor 0 de resultado. Assim:

$$\begin{array}{r} A = 00111010 \\ B = 01111100 \\ \hline 01000110 \end{array}$$

TIPO: XOR data

EXEMPLO: XOR # 21

FUNÇÃO: Esta instrução quer dizer "XOR o valor do byte seguinte com o conteúdo do Acumulador, guardando neste o resultado".

FLAGS: C-0 Z-X S-X P/O-X AC-1 N-0

TIPO: XOR reg

EXEMPLO: XOR H

FUNÇÃO: Esta instrução quer dizer "XOR o conteúdo do registo reg com o conteúdo do Acumulador, guardando neste o resultado".

FLAGS: C-0 Z-X S-X P/O-P AC-1 N-0

TIPO: XOR (HL)

XOR (indreg + d)

EXEMPLO: XOR (IY + 40)

FUNÇÃO: Esta instrução significa "XOR o conteúdo do byte endereçado por HL (ou indreg + d) com o conteúdo do Acumulador, colocando o resultado neste".

FLAGS: C-0 Z-X S-X P/O-P AC-1 N-0

Importa tecer alguns comentários a este grupo de instruções muito usado.

É habitual usar-se AND A antes de operações aritméticas com a Carry, pois o valor de A não é alterado (pode verificar) e a Carry é posta a zero.

Outra utilidade reside no movimento de gráficos de alta resolução ou na identificação de uma gama limitada de códigos.

No próximo capítulo examinaremos as instruções de salto.

## CAPÍTULO 8

### INSTRUÇÕES DE SALTO

Em BASIC, dispomos de várias instruções (GO TO, GO SUB, IF... THEN) que nos permitem alterar a sequência crescente normal da execução de um programa. Em Assembler possuímos instruções que fazem algo similar — as instruções de salto JR, JP e CALL. Estas instruções podem, por sua vez, estar sujeitas a condições (o estado de uma certa flag), equivalendo a IF... THEN GO TO / GO SUB.

#### Grupo JR

JR significa “Jump Relative”, isto é, salto relativo. Esta instrução é muito usada, pois é bastante rápida e económica, acrescentando-se ainda a vantagem de não usar endereços absolutos. Os programas ditos relocatáveis (funcionam em qualquer local da memória) usam estas instruções para os saltos (e também os CALL). Em contrapartida, os saltos que esta instrução pode realizar estão limitados de - 126 a + 129 byte desde o endereço do código da instrução. O Contador de Programa ajusta-se automaticamente.

TIPO: JR d

EXEMPLO: JR # 32

**FUNÇÃO:** Esta instrução significa "salte para a instrução indicada pelo deslocamento em relação à instrução". Na prática, isto não se faz assim, usamos nomes: JR LOOP. Evitamos assim de contar byte.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** JR cond,d

**EXEMPLO:** JR NC, # 80  
JR Z,LOOP2

**FUNÇÃO:** Esta instrução significa "salte para a instrução indicada pelo deslocamento em relação à instrução se a condição cond for verificada; de outro modo ignore esta instrução". De novo chamo a atenção para o uso de nomes, que facilita o trabalho. Note que as condições aceitáveis pela instrução são C, NC, Z e NZ.

**FLAGS:** Não são afectadas por este tipo de instrução.

#### Grupo JP

Este grupo diferencia-se do anterior por 3 razões: o salto não é limitado (pode endereçar-se a qualquer byte da memória), usa mais 1 byte (com a excepção de JP (HL) e JP (indreg)) e é mais rápida (não há necessidade de intemamente se calcularem deslocamentos); por isto se chama salto absoluto. Os programas relocatáveis não podem, assim, usar este grupo de instruções.

**TIPO:** JP nome

**EXEMPLO:** JP GRAFIC

**FUNÇÃO:** Esta instrução é lida como "salte para o byte endereçado pelo nome "nome"".

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** JP (HL)

JP (indreg)

**EXEMPLO:** JP (IX)

**FUNÇÃO:** Esta instrução quer dizer "salte para o byte endereçado pelo conteúdo de HL (ou indreg)".

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** JP cond, nome

**EXEMPLO:** JP PO,VERIFI

**FUNÇÃO:** Esta instrução significa "salte para o byte endereçado pelo nome "nome" se a condição for verificada; em caso contrário, ignore a instrução". Note que as condições aceitáveis são C, NC, Z, NZ, P, M, PO e PE.

**FLAGS:** Não são afectadas por este tipo de instrução.

#### Grupo CALL

Este grupo é formado por instruções muito semelhantes ao GO SUB do BASIC. Essencialmente, quando o PC encontra uma instrução CALL (executável, no caso de ser condicional), guarda o endereço da instrução seguinte no SP e salta para a subrotina chamada. A saída da subrotina é realizada pela instrução RET (de RETURN), que pode ser condicional ou não, reavendo o PC o endereço guardado no SP, resumindo a sequência do programa. Importa chamar a atenção para o uso correcto de subrotinas em código de máquina, pois um mau planeamento do programa impedirá de fazer o que deseja. Por isto, veremos adiante como proceder para evitar erros comuns no uso desta opção.

**TIPO:** CALL nome

**EXEMPLO:** CALL INÍCIO

**FUNÇÃO:** Esta instrução quer dizer "chame a subrotina que se inicia em "nome"".

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** CALL cond, nome

**EXEMPLO:** CALL C,AJUSTA

**FUNÇÃO:** Esta instrução significa "chame a subrotina que se inicia em "nome" se a condição for verificada; caso contrário, execute a instrução seguinte". Note que as condições são C, NC, Z, NZ, P, M, PO e PE.

**FLAGS:** Não são afectadas por este tipo de instrução.

TIPO: RET

FUNÇÃO: Esta instrução significa “retorne da subrotina”  
Note que deve haver sempre um número de RET igual ao de CALL, de modo a que não ocorram erros no SP.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: RET cond

EXEMPLO: RET NC

FUNÇÃO: Esta instrução quer dizer “retorne da subrotina se a condição for verificada; caso contrário, ignore-a e processe a instrução seguinte”.

FLAGS: Não são afectadas por este tipo de instrução.

Veremos em seguida como se criam loops em Assembler, onde encontraremos uma das instruções mais importantes e poderosas — DJNZ.

## CAPÍTULO 9

### LOOPS

Inevitavelmente, haverá uma altura em que queremos fazer operações iterativas, isto é, queremos repetir várias vezes uma mesma sequência de instruções. Para tal usam-se os LOOPS.

Em BASIC, a sequência habitual é do estilo

```
10 FOR N= 1 TO 20  
.....  
.....  
.....  
100 NEXT N
```

Podemos fazer algo semelhante em Assembler, mas gasta-se muito tempo e byte. Podemos poupar em ambos os lados se usarmos uma contagem decrescente. O equivalente do BASIC seria do estilo

```

10 FOR N= 20 TO 1 STEP - 1
.....
....
...
100 NEXT N

```

Sucedee que não temos uma mnemónica do tipo FOR...NEXT. Cada vez que queremos fazer um loop, teremos de usar um registo ou o conteúdo de um byte como contador de iterações e testá-lo em cada passagem do loop. Felizmente, a instrução DJNZ substitui a sequência decrementa-e-segue-se-condição.

Esta instrução usa o registo B como contador. Cada vez que o PC a executa, sucede o seguinte:

- 1) Decrementa B
- 2) Se o conteúdo de B que ficar não for zero, salta para o início do loop, que estará entre -126 a + 129 byte desta instrução.
- 3) Se o conteúdo de B for zero após a decrementação, a instrução seguinte será executada.

Exemplificando:

```

..... ;PROGRAMA
LD B,# 06 ;LOOP REPETIDO 6 VEZES
LOOP      ; INÍCIO DO LOOP
.....
.....
DJNZ, LOOP
..... ; CONTINUAÇÃO DO PROGRAMA

```

Já deve ter reparado que, pelo facto de se usar um registo, o loop com DJNZ está limitado a 256 iterações. Para maior número, podemos recorrer a duas soluções: a) simulamos a instrução DJNZ, usando um par de registos em vez do registo B; b) usamos dois loops DJNZ ou mesmo mais (sendo este método mais complicado que o primeiro e mais propenso a erro). Exemplifiquemos a estrutura de um loop com um controlador de 16-bit:

```

..... ; PROGRAMA
LD HL,# NNNN ; LOOP REPETIDO # NNNN VEZES
.... LOOP ; INÍCIO DO LOOP
.....
.....
DEC HL ; SIMULAÇÃO DE DJNZ PARA 16-BIT
LD A,# 00 ; (PARTINDO DO PRINCÍPIO QUE
; A NÃO É
; UTILIZADO DENTRO DO LOOP OU
; QUE O
; SEU VALOR NÃO NECESSITA DE SER
; MANTIDO)
CP H ; TESTE PARA O MSB
JR NZ, LOOP; H DIFERENTE DE ZERO: AINDA HÁ
; ITERAÇÕES
CP L ; TESTE PARA O LSB, POIS H= 0
JR NZ, LOOP; L DIFERENTE DE ZERO: AINDA HÁ
; ITERAÇÕES A FAZER
..... ; CONTINUAÇÃO DO PROGRAMA. HL= 0

```

Vejamos agora um loop análogo, mas usando a instrução DJNZ:

```

..... ; PROGRAMA
LD A,# 00 ; INICIALIZAÇÃO DO ACUMULADOR
LD C,# NN ; LOOP REPETIDO# NN * 256 VEZES
LOOP1 LD B,# 00 ; PARA 256 VEZES
LOOP  ; INÍCIO DO LOOP
.....
.....
DJNZ LOOP
DEC C
CP C ; C= 0 ? SE SIM SET FLAG Z
JR NZ LOOP1
..... ; CONTINUAÇÃO DO PROGRAMA

```

Note que para obtermos loops não múltiplos de 256 a estrutura complica-se:

```
..... ; PROGRAMA
LD A,# 00 ; INICIALIZAÇÃO DO ACUMULADOR
LD C,# NN ; LOOP REPETIDO #NN * 256 VEZES
LD B,# NN ; PRIMEIRO LOOP
JR LOOP
LD B,# 00 ; PARA 256 VEZES
..... ; INÍCIO DO LOOP
.....
.....
DJNZ LOOP
DEC C
CP C ; C= 0 ? SE SIM SET FLAG Z
JR NZ LOOP1
..... ; CONTINUAÇÃO DO PROGRAMA
```

Todas as instruções que usei foram já explicadas. Se o exemplo lhe pareceu difícil, veja o significado das instruções que lhe pareceram obscuras. Note que se pressupõe que o loop está entre a faixa de +129 byte de qualquer dos JRs usados.

Veremos de seguida um outro grupo de instruções muito poderosas e úteis: as de transferência e pesquisa de blocos.

## CAPÍTULO 10

### TRANSFERÊNCIA E PESQUISA DE BLOCOS

Por vezes necessitamos de mover um número razoável de conteúdos de byte ordenados, de uma zona da memória para outra, por exemplo: temos um écran a partir de # 9C40 e queremos transferi-lo para a zona apropriada (# 4000, se não sabia). Outras vezes queremos ver se existe um (ou mais) dado valor numa série de byte, ou comparar duas séries de byte para ver se são iguais.

O Z80 possui instruções para estes fins e que serão o tema deste capítulo.

#### Grupo LD

Estas instruções usam os pares HL, DE e BC.

#### INSTRUÇÃO: LDD

**FUNÇÃO:** Esta instrução significa “transfira o conteúdo de um byte, endereçado por HL, para o byte cujo endereço é dado pelo conteúdo do par DE. Decrementa o conteúdo dos pares BC, DE e HL”.

EXEMPLO: HL = # 9C40 DE = # 4000 BC = # 1B00  
byte # 9C40 = # 05

Após a execução da instrução LDD: HL = # 9C3F BC = # 1AFF  
DE = # 3FFF  
byte # 4000 = # 05

FLAGS: C- NA Z- NA AC- 0 N- 0

A P/O é 1 (set) se BC- 1 for diferente de Zero, 0 (reset) em todos os outros casos.

### INSTRUÇÃO: LDDR

FUNÇÃO: Esta instrução significa “transfira o conteúdo de um byte, endereçado por HL, para o byte cujo endereço é dado pelo conteúdo do par DE. Decrementa o conteúdo dos pares BC, DE e HL, até BC = 0.” (Note que após cada transferência, os interruptores são reconhecidos e inclui 2 ciclos de refrescamento).

EXEMPLO: HL = # 9C40 DE = # 4000 BC = # 0001  
byte # 9C40 = # 05

Após a execução da instrução LDDR: HL = # 9C3F BC = # 0000  
DE = # 3FFF  
byte # 4000 = # 05

FLAGS: C- NA Z- NA S- NA AC- 0 N- 0

A P/O é 1 (set) se BC- 1 for diferente de Zero, 0 (reset) em todos os outros casos.

### INSTRUÇÃO: LDI

FUNÇÃO: Esta instrução significa “transfira o conteúdo de um byte, endereçado por HL, para o byte cujo endereço é dado pelo conteúdo do par DE. Incrementa o conteúdo dos pares DE e HL. Decrementa o conteúdo do par BC”.

EXEMPLO: HL = # 9C40 DE = # 4000 BC = # 1B00  
byte # 9C40 = # 05

Após a execução da instrução LDI: HL = # 9C41 BC = # 1AFF  
DE = # 4001  
byte # 4000 = # 05

FLAGS: C- NA Z- NA S- NA AC- 0 N- 0

A P/O é 1 (set) se BC- 1 for diferente de Zero, 0 (reset) em todos os outros casos.

### INSTRUÇÃO: LDIR

FUNÇÃO: Esta instrução significa “transfira o conteúdo de um byte, endereçado por HL, para o byte cujo endereço é dado pelo conteúdo do par DE. Incrementa o conteúdo dos pares DE e HL. Decrementa o conteúdo do par BC. Repete até BC = 0”. Note que esta instrução resolveria a questão da transferência do écran.

EXEMPLO: HL = # 9C40 DE = # 4000 BC = # 1B00  
Após a execução da instrução LDIR: HL = # AE3F BC = # 0000  
DE = # 5AFF

O écran estaria transferido.

FLAGS: C- NA Z- NA S- NA AC- 0 N- 0

A P/O é 1 (set) se BC- 1 for diferente de Zero, 0 (reset) em todos os outros casos.

### Grupo CP

Analogamente, este grupo realiza a operação de comparação usando os pares HL e BC, testando com o conteúdo de A.

### INSTRUÇÃO: CPI

FUNÇÃO: Esta instrução compara o conteúdo do Acumulador com o conteúdo do byte endereçado por HL. Incrementa HL e decrementa BC.



EXEMPLO: HL= # 6000 BC= # 0143 A= # 08  
byte # 60000= #08

Após a execução da instrução CPI: HL= #6001 BC= #0142  
A= # 08

As flags darão: S= 0, AC= 0, P / O= 1, N=1 e Z= 1.

FLAGS: C- NA Z- X S- X P / O- X AC- X N- 1

#### INSTRUÇÃO: CPIR

FUNÇÃO: Esta instrução compara o conteúdo do Acumulador com o conteúdo do byte endereçado por HL. Incrementa HL e decrementa BC. Repete até BC= 0 ou até A= (HL).

FLAGS: C- NA Z- X S- X P/O- X AC- X N- 1

#### INSTRUÇÃO: CPD

FUNÇÃO: Esta instrução compara o conteúdo do Acumulador com o conteúdo do byte endereçado por HL. Decrementa HL e BC.

EXEMPLO: HL= # 6000 BC= # 0143 A= # 08  
byte # 6000= # 08

Após a execução da instrução CPI: HL= # 5FFF BC= #0142  
A= # 08

As flags: S= 0, P /O= 1, N= 1 e Z= 1.

FLAGS: C- NA Z- X S- X P/O- X AC- X N- 1

#### INSTRUÇÃO: CPDR

FUNÇÃO: Esta instrução compara o conteúdo do Acumulador com o conteúdo do byte endereçado por HL. Decrementa HL e BC. Repete até BC= 0 ou até A= (HL).

FLAGS: C- NA Z- X S- X P/ O- X N- 1

Com esta instrução terminamos este capítulo. Para pesquisar ou deslocar um grande número de byte, estas instruções são de grande utilidade. O capítulo seguinte trata de operações com o Stack.

## CAPÍTULO 11

### OPERAÇÕES DE STACK

Ao longo destas páginas já tivemos oportunidade de referir as vantagens de dispormos de uma "pilha", que o Z80 suporta: o STACK. Se consultar a página 165 do seu manual do Spectrum encontrará uma coisa chamada "machine stack". O stack permite que o Z80 guarde informação, sem ter de se recordar do endereço onde ela está.

O stack é, primariamente, um armazém de endereços e, como tal, a sua unidade de informação é de 16-bit. Outro aspecto a considerar é que o stack cresce para baixo; novas entradas no stack são colocadas em cima e as saídas são também feitas por cima. Isto implica que, para retirarmos informação, devemos respeitar a regra do "último entrado-primeiro a sair", sem o que incorrer-se-á em situação de erro. Note que o stack é automaticamente usado por algumas instruções (ex: CALL).

Qual a dimensão que devemos reservar na memória para o stack? Geralmente, um bloco de 256 byte é suficiente para qualquer tipo de aplicação. Se o programa for muito longo, devemos ter o cuidado de inicializar o stack num local conveniente, com a instrução LD SP,data16. Note que o sistema operativo do Spectrum coloca o stack abaixo do RAMTOP, na área de BASIC.

Por outro lado, não se esqueça de que, se inicializar o stack para um local diferente de memória e após utilizar o seu programa em máquina retornar ao sistema operativo normal do Spectrum, deverá restituir o Stack à sua posição original. A sequência é do tipo:

```
LD (end), SP : GUARDAR POSIÇÃO ORIGINAL DO STACK
LD SP, data16 ;INICIALIZAR O STACK NO LOCAL
               ;ESCOLHIDO
.....       ;PROGRAMA
.....
.....
```

```
LD HL,(end);HL= POSIÇÃO ORIGINAL DO STACK
LD SP, HL  ;RESTITUIR O VALOR ORIGINAL DO STACK
RET        ;RETORNO AO BASIC
```

TIPO: PUSH pareg	EXEMPLO: PUSH AF
PUSH indreg	PUSH IX

**FUNÇÃO:** Esta instrução significa "coloque o conteúdo de pareg (ou indreg) no topo do stack". Note que PUSH pareg necessita de 1 byte e 11 ciclos, ao passo que PUSH indreg precisa de 2 byte e 15 ciclos.

**FLAGS:** Não são afectadas por este tipo de instrução.

TIPO: POP	EXEMPLO: POP BC
POP indreg	POP IX

**FUNÇÃO:** Esta instrução significa "coloque o conteúdo dos dois byte do topo do stack no pareg (ou indreg) especificado". O comentário da instrução anterior a respeito de ciclos de relógio e número de byte necessário é aqui igualmente relevante. Note que pode fazer com o stack o equivalente a LD pareg, pareg, instrução que não existe. EX: PUSH HL seguido de POP BC.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: EX (SP),HL

EX (SP), indreg

EXEMPLO: EX (SP),IX

FUNÇÃO: Esta instrução quer dizer “troque o conteúdo do topo do stack com o conteúdo de HL (ou indreg)”. Note que a instrução que usa HL necessita de 1 byte e 19 ciclos, ao passo que a que usa indreg precisa de 2 byte e 23 ciclos.

FLAGS: Não são afectadas por este tipo de instrução.

Note que, no seu programa, o número de instrução POP deverá ser igual ao número de instruções PUSH. De outro modo ocorrerão erros ou um “crash”. Não use o stack para passar parâmetros para subrotinas!

O capítulo seguinte tratará de trocas.

## CAPÍTULO 12

### TROCAS

Por vezes, necessitamos de preservar os valores dos registos ou das flags, enquanto se efectuam certas operações. Para este fim, podemos usar os registos e flags alternativas de que já falámos.

TIPO: EX AF, AF'

FUNÇÃO: Esta instrução significa “mude para o par AF alternativo (troque o conteúdo do par AF com o conteúdo do par AF)”.

FLAGS: São também trocadas (F).

TIPO: EXX

FUNÇÃO: Esta instrução quer dizer “troque os pares BC, DE e HL com os pares alternativos BC', DE' e HL”.

FLAGS: Não são afectadas por este tipo de instrução.

TIPO: EX DE,HL

**FUNÇÃO:** Esta instrução significa “troque o conteúdo de DE com o conteúdo de HL”.

**FLAGS:** Não são afectadas por este tipo de instrução.

Temos ainda trocas baseadas no conteúdo do SP.

**TIPO:** EX (SP),HL

EX (SP),indreg

**EXEMPLO:** EX (SP),IX

**FUNÇÃO:** Esta instrução significa “troque o conteúdo dos byte endereçados por SP com o conteúdo de HL (ou indreg)”.

**FLAGS:** Não são afectadas por este tipo de instrução.

O próximo capítulo tratará das instruções de bit.

## CAPÍTULO 13

### BITS

Em certas ocasiões é do nosso interesse examinar o estado de um bit ou modificá-lo de acordo com as necessidades de um programa. Para este fim, o Z80 contém algumas instruções de manipulação de bit.

**TIPO:** BIT b,reg

**EXEMPLO:** BIT 5,A

**FUNÇÃO:** Esta instrução equivale à pergunta “o bit b do registo reg é igual a 1 ou igual a zero? Coloque o seu complemento na flag Z”. Supondo que A = 00011010B, Z será, no nosso exemplo, igual a zero (reset).

**FLAGS:** C- NA Z- X S- ? P / O- ? AC- 1 N- 0

**TIPO:** BIT b,(HL)

BIT b,(indreg + d)

**EXEMPLO:** BIT 3,(HL)

BIT 2,(IX + 07)

**FUNÇÃO:** Esta instrução significa “coloque o complemento do bit b do byte endereçado por HL (ou por  $\text{indreg} + d$ ) na flag Z”. Note que a instrução que recorre a HL usa apenas 2 byte e 12 ciclos, necessitando a que usa registos de indexamento de 4 byte e 20 ciclos.

**FLAGS:** C- NA Z- X S- ? P / O- ? AC- 1 N- 0

**TIPO:** SET b, reg

**EXEMPLO:** SET 4, D

**FUNÇÃO:** Esta instrução quer dizer “ponha o bit b do registo reg com o valor 1”.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** SET b, (HL)  
SET b, ( $\text{indreg} + d$ )

**EXEMPLO:** SET 1, (HL)  
SET 7, ( $\text{IX} + 18$ )

**FUNÇÃO:** Esta instrução significa “ponha o bit b do byte endereçado por HL (ou por  $\text{indreg} + d$ ) com o valor 1”. Note que a instrução que usa HL precisa de 2 byte e 15 ciclos, ao passo que a que usa registos de indexamento necessita de 4 byte e 23 ciclos.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** RES b, reg

**EXEMPLO:** RES 6, L

**FUNÇÃO:** Esta instrução significa “ponha o bit b do registo reg com o valor 0”. É a instrução inversa de SET b, reg.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** RES b, (HL)  
RES b, ( $\text{Indreg} + d$ )

**EXEMPLO:** RES 2, (HL)  
RES 1, ( $\text{IX} + 80$ )

**FUNÇÃO:** Esta instrução quer dizer “ponha o bit b do byte endereçado por HL (ou por  $\text{indreg} + d$ ) com o valor 0”. Os comentários tecidos acerca de SET b, (HL) e SET b, ( $\text{indreg} + d$ )

sobre os byte necessários e ciclos de relógio têm aqui igual validade.

**FLAGS:** Não são afectadas por este tipo de instrução.

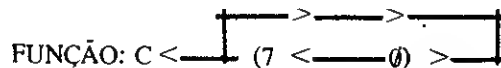
Veremos agora um grupo mais extenso, o das rotações.

## CAPÍTULO 14

### ROTAÇÕES E MUDANÇAS

Embora constituindo um grupo útil, as instruções que vamos ver são geralmente pouco usadas. Na minha opinião, o pouco uso que têm deve-se mais a uma dificuldade em as distinguir claramente, do que a quaisquer outros motivos. Através destas instruções podemos obter algumas operações simples de multiplicação e divisão por 2 e apresentações gráficas suaves no seu movimento. Algumas destas instruções usam a CARRY como um nono bit (bit 8; se usarmos a convenção de numeração de bit num byte de 0 a 7).

TIPO: RLCA

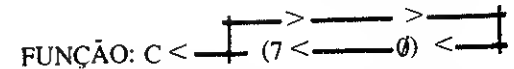


Graficamente, esta é a representação de RLCA (Rotate Accumulator Left Circular). Esta instrução roda o conteúdo do Acumulador para a esquerda de um bit, copiando o bit 7 na Carry. Suponha que A = 01111010 B = 1. Após a execução da instrução RLCA teremos A = 11110100 b e C = 0. RLCA deve ser usada como instrução lógica.

FLAGS: C- X Z- NA S- NA P / O- NA AC- 0 N- 0

TIPO: RLC reg

EXEMPLO: RLC D



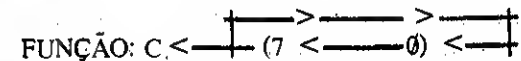
Graficamente, esta é a representação de RLC reg (Rotate Contents of Register Left Circular). Esta instrução roda o conteúdo do Acumulador para a esquerda de um bit, copiando o bit 7 na Carry. Suponha que D = 01100010 B e C = 1. Após a execução da instrução RLCA teremos A = 11000101 B e C = 0. RLC reg deve ser usado como instrução lógica.

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: RLC (HL)

RLC (indreg + d)

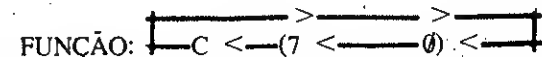
EXEMPLO: RLC (IX + 04)



Como podemos ver, este tipo de instrução é similar às duas anteriores, mas usando endereçamento indirecto ao byte em questão, a partir de HL ou de um registo de indexamento mais deslocado. Note-se que a instrução que recorre a HL necessita de 2 byte e 15 ciclos ao passo que as que usam indreg + d precisam de 4 byte e 23 ciclos de relógio.

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: RLA



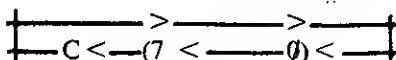
Como se pode ver pelo desenho, esta instrução roda o conteúdo do Acumulador para a esquerda de um bit, através do estado da

Carry. RLA é a mnemônica de "Rotate Acumulador Left Through Carry". Suponha que A = 0111010 B e C = 1. Após a execução de RLA teremos A = 1110101 e C = 0. Repare nas diferenças importantes que ocorrem neste grupo de instruções tão semelhantes, aparentemente.

FLAGS: C- X Z- NA S- NA P /O- NA AC- 0 N- 0

TIPO: RL reg

EXEMPLO: RL B

FUNÇÃO: 

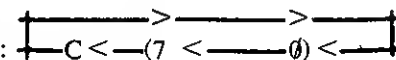
Instrução similar a RLA para os outros registros.

FLAGS: C- X Z- X S- X P /O- P AC- 0 N- 0

TIPO: RL (HL)

RL (indreg + d)

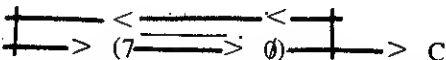
EXEMPLO: RL (IY + 55)

FUNÇÃO: 

De novo, esta instrução é similar a RLA, mas considerando endereçamento indirecto. Note que usando HL precisamos de 2 byte e 15 ciclos de relógio, enquanto que indreg + d usa 4 byte e 23 ciclos.

FLAGS: C- X Z- X S- X P /O- P AC- 0 N- 0

TIPO: RRCA

FUNÇÃO: 

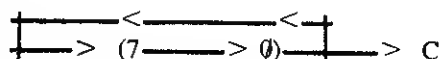
Esta é uma instrução simétrica de RLCA, pois roda o conteúdo do Acumulador de um bit para a direita, copiando o bit 0 no estado

da Carry. Supondo que A = 10001010 e C = 1, após a execução de RRCA teremos A = 01000101 B e C = 0.

FLAGS: C- X Z- NA S- NA P /O- NA AC- 0 N- 0

TIPO: RRC reg

EXEMPLO: RRC C

FUNÇÃO: 

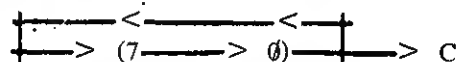
Analogamente, esta instrução equivale a RRCA para os restantes registos e é simétrica a RLC reg.

FLAGS: C- X Z- X S- X P /O- P AC- 0 N- 0

TIPO: RRC (HL)

RRC (indreg + d)

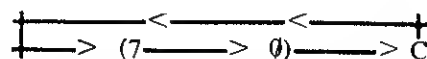
EXEMPLO: RRC (IY + 145)

FUNÇÃO: 

Esta instrução pode ser descrita com as mesmas palavras de RRCA, considerando que o byte a ser operado é endereçado indirectamente por HL ou indreg + d. Note que, de modo semelhante a RCL (HL), RRC (HL) usa 2 byte e 15 ciclos e RRC (indreg + d) gasta 4 byte e 23 ciclos.

FLAGS: C- X Z- X S- X P /O- P AC- 0 N- 0

TIPO: RRA

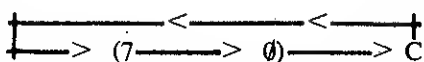
FUNÇÃO: 

Esta instrução é simétrica de RLA. RRA significa "rode o conteúdo do Acumulador para a direita de um bit através do estado da Carry".

FLAGS: C- X Z- NA S- NA P /O- NA AC- 0 N- 0

TIPO: RR reg

EXEMPLO: RR L

FUNÇÃO: 

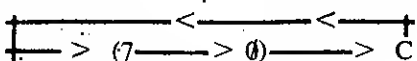
Esta instrução é análoga a RRA, considerando os outros registros, sendo ainda simétrica a RL reg.

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: RR (HL)

RR (indreg + d)

EXEMPLO: RR (IX + 12)

FUNÇÃO: 

Simétrica a RL (HL) e RL (indreg + d), esta instrução desempenha a mesma função, mas rodando para a direita o byte indirectamente endereçado. Também aqui RR (HL) usa 2 byte e 15 ciclos, ao passo que RR (indreg + d) necessita de 4 byte e 23 ciclos.

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: SLA reg

EXEMPLO: SLA B

FUNÇÃO:  $C \leftarrow (7 \leftarrow \text{---} 0) \leftarrow 0$

Esta instrução significa "desloque para a esquerda o conteúdo do registo reg, colocando zero no bit de ordem inferior". A mnemónica corresponde ao inglês "Shift contents of register Left Arithmetic". Efectivamente, SLA reg equivale a dobrar o conteúdo de reg (multiplicar por 2).

FLAGS: C- X Z- X S- X P / O- X AC- 0 N- 0

TIPO: SLA (HL)

SLA (indreg + d)

EXEMPLO: SLA (IX + 31)

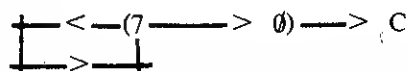
FUNÇÃO:  $C \leftarrow (7 \leftarrow \text{---} 0) \leftarrow 0$

Esta instrução desempenha as mesmas funções de SLA reg, para byte endereçado indirectamente. Note que SLA (HL) usa 2 byte e 15 ciclos, enquanto que SLA (indreg + d) necessita de 4 byte e 23 ciclos.

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: SRA reg

EXEMPLO: SRA A

FUNÇÃO: 

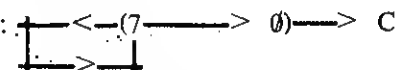
Esta instrução significa "desloque o registo reg um bit para a direita. Preserve o valor do bit mais significativo". Note que, apesar de ser um "shift" aritmético SRA reg NÃO É a instrução simétrica de SLA reg. Esta instrução tem utilidade quando usamos explicitamente a notação numérica com números positivos e negativos. SRA reg efectua a divisão por 2, mas preserva o sinal do número (bit 7).

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: SRA (HL)

SRA (indreg + d)

EXEMPLO: SRA (IX + 99)

FUNÇÃO: 

Esta instrução é análoga a SRA reg, usando endereçamento indirecto ao byte a ser operado. Note que usando HL necessita de 2 byte e 15 ciclos de relógio, enquanto que indreg + d usa 4 byte e 23 ciclos.

FLAGS: C- X Z- X S- X P / O- P AC- 0 N- 0

TIPO: SRL reg

EXEMPLO: SRL H

FUNÇÃO:  $0 \rightarrow (7 \text{---} 0) \rightarrow C$



Esta é a instrução simétrica a SLA reg. Note que realiza a divisão por 2 do conteúdo do registo especificado, mas não considera o seu sinal (efectivamente, o resultado é sempre positivo). É a mnemónica correspondente a "Shift contents of register Right Logically".

FLAGS: C- X Z- X S- X P/O- P AC- 0 N- 0

TIPO: SRL (HL)

SRL (indreg + d)

EXEMPLO: SRL (IX + 0)

FUNÇÃO: 0--- > (7----- > 0)--- > C

Esta instrução efectua a mesma operação da anterior, com endereçamento indirecto ao byte a ser operado. Também aqui SRL (HL) usa 2 byte e 15 ciclos e SRL (indreg + d) necessita de 4 byte e 23 ciclos.

FLAGS: C- X Z- X S- X P/O- P AC- 0 N- 0

Há ainda mais duas instruções de rotação, mas referi-las-emos quanto tratarmos de notação BCD (Binary-Coded-Decimal), que é uma forma de representar informação em formato decimal.

O próximo capítulo referirá as instruções que o Z80 dispõe para comunicar com o exterior.

## CAPÍTULO 15

### O MUNDO EXTERIOR — IN E OUT

O Z80 nunca se desloca, pois é muito comodista. Ele espera que as informações que necessita venham por uma das suas portas e por elas envia os dados que trata. Na realidade, o Z80 tem 256 portas (em inglês será "port", isto é, porto). No entanto, a configuração do hardware determina aquelas que são disponíveis.

O Spectrum, do modo como está concebido, permite-nos aces- so a apenas duas portas: #FB e #FE; a primeira controla os contactos com a impressora e a segunda diz respeito à leitura do teclado e ao gravador (LOAD, SAVE, MERGE e VERIFY).

Sempre que o Z80 necessita de saber qualquer coisa, terá de ser executada uma instrução IN; reciprocamente, quando tem algo a enviar para fora, recorre a instruções OUT. A noção por detrás destes grupos de instruções é similar às funções IN e OUT do BASIC (pág. 159 em diante do Manual).

#### GRUPO IN

TIPO: IN A, (porta)

EXEMPLO: IN A, (#FE)

**FUNÇÃO:** Esta instrução significa "coloque no Acumulador um byte de data a partir da porta I / O especificada". No caso do exemplo, a execução faria a leitura do teclado do Spectrum.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** IN reg.(C)                      **EXEMPLO:** IN B,(C)

**FUNÇÃO:** Esta instrução quer dizer "coloque no registo reg um byte de data proveniente da porta I / O especificada pelo conteúdo do registo C". Uma particularidade desta instrução: se o segundo byte for #70, só as flags serão afectadas.

**FLAGS:** C- NA Z- X S- X P/O- P AC- X N- 0

**TIPO:** IND

**FUNÇÃO:** Esta instrução quer dizer "receba um input do porto I / O, especificado pelo conteúdo do registo C, e coloque-o no local de memória endereçado pelo conteúdo de HL. Decrementa B e HL". Note que é equivalente a LDD para portos I / O.

**FLAGS:** C- NA Z- X S- ? P/O- ? AC- ? N- 1

**TIPO:** INDR

**FUNÇÃO:** Esta instrução significa "transfira um bloco de data do porto I / O endereçado pelo conteúdo do registo C para o local de memória endereçado pelo conteúdo de HL, indo de endereços mais altos para os mais baixos. O conteúdo de B é decrementado por cada transferência até B= 0". Note que a instrução de bloco equivalente a LDDR para portos I / O.

**FLAGS:** C- NA Z- X S- ? P/O- ? AC- ? N- 1

**TIPO:** INI

**FUNÇÃO:** Esta instrução é lida como "transfira um byte de data do porto I / O endereçado pelo conteúdo do registo C para o

local de memória endereçado pelo conteúdo de HL. Decrementa B e incrementa HL". Instrução equivalente a LDI para portos I / O.

**FLAGS:** C- NA Z- X S- ? P/O- ? AC- ? N- 1

**TIPO:** INIR

**FUNÇÃO:** Esta instrução significa "transfira um bloco de data do porto I / O endereçado pelo conteúdo do registo C para o local de memória endereçado pelo conteúdo de HL. Decrementa B e incrementa HL. Repita até B= 0". Instrução equivalente a LDIR para portos I / O.

**FLAGS:** C- NA Z- X S- ? P/O- ? AC- ? N- 1

## GRUPO OUT

**TIPO:** OUT (porto), A                      **EXEMPLO:** OUT (#FB),A

**FUNÇÃO:** Esta instrução significa "envie o conteúdo do Acumulador para o porto I / O especificado". No exemplo, o conteúdo do Acumulador seria enviado para o buffer da impressora.

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** OUT (C),reg

**FUNÇÃO:** Esta instrução quer dizer "envie o conteúdo do registo reg para o porto I / O especificado pelo conteúdo do registo C".

**FLAGS:** Não são afectadas por este tipo de instrução.

**TIPO:** OUTD

**FUNÇÃO:** Esta instrução significa "envie o conteúdo do local de memória endereçado por HL para o porto I / O endereçado pelo registo C. Decrementa os registos B e HL". É o oposto de IND.

FLAGS: C- NA Z- X S- ? P/O- ? AC- ? N- 1

TIPO: OTDR

**FUNÇÃO:** Esta instrução significa “envie o conteúdo do local de memória endereçado por HL para o porto I / O endereçado pelo registo C. Decrementa os registos B e HL. Repita até B = 0”. Note que é o oposto de INDR.

FLAGS: C- NA Z- X S- ? P/O- ? AC- ? N- 1

TIPO: OUTI

**FUNÇÃO:** Esta instrução significa “envie o conteúdo do local de memória endereçado por HL para o porto I / O endereçado pelo registo C. Decrementa o registo B e incrementa HL”. Note que é o oposto de INI.

FLAGS: C- NA Z- X S- ? P/O- ? AC- ? N- 1

TIPO: OTIR

**FUNÇÃO:** Esta instrução significa “envie o conteúdo do local de memória endereçado por HL para o porto I / O endereçado pelo registo C. Decrementa o registo B e incrementa HL. Repita até B = 0”. É o oposto de INIR.

FLAGS: C- NA Z- X S- ? P/O- ? AC- ? N- 1

O próximo capítulo tratará as instruções de interrupção.

## CAPÍTULO 16

### INTERRUPÇÕES

O Z80 examina inputs como parte de cada ciclo de instrução, que dão pelo nome inglês de “INTERRUPTS”, ou seja, interrupções. Basicamente, um interrupt é um input enviado ao microprocessador, que pode ocorrer a qualquer momento e que geralmente suspende a execução do programa, sem este saber. No caso do Spectrum, o programa é interrompido 50 vezes por segundo, para a leitura do teclado, pela rotina da ROM que desempenha tal função.

Quando um interrupt é aceite, o conteúdo do PC é colocado no stack. Seguidamente, a rotina que provocou o interrupt coloca no PC o endereço da rotina substituta — a rotina de interrupção.

Um interrupt pode ser permitido ou não. Veremos as instruções apropriadas para o fazer. No entanto, há interrupts que não podem ser desligados — são conhecidos por “non-maskable interrupt”. Não nos deteremos na descrição da resposta interna do Z80 aos interrupts (que é algo complexa) e que varia conforme o MODO presente de interrupção.

O Z80 tem 3 modos de interrupção e é por aqui que vamos começar.

## MODO 0

### TIPO: IM 0

**FUNÇÃO:** Esta instrução coloca o Z80 no modo 0 de interrupt. Neste modo, o interrupt coloca uma instrução no DATA BUS, que o Z80 executará. Não afecta registos ou flags.

O input de data é feito a partir de instruções RST (que adiante veremos).

## MODO 1

### TIPO: IM 1

**FUNÇÃO:** Neste modo, o Z80 executa sempre um salto para RST # 38 (ou RST 56, decimal). É equivalente ao anterior se o input de data for RST # 38. Este é o modo que o Spectrum usa normalmente. Não afecta registos ou flags.

## MODO 2

### TIPO: IM 2

**FUNÇÃO:** Este modo é mais complicado, mas permite-nos fazer uso de interrupts. No modo 2, o Z80 usa o input de data como parte do endereço inicial da rotina alternativa (rotina de interrupt). Como, no caso do Spectrum, não fornece este byte de input, o Z80 "pensa" que o seu valor é # FF ou 255 decimal. Este é o byte de ordem inferior e é, portanto, fixo. O byte de ordem superior é recolhido do valor do registo I. No entanto, mexer no conteúdo do registo I pode trazer problemas, nomeadamente interferências no écran. O único processo de evitar isto é recolhermos esse byte (chamado de vector) na ROM. Em apêndice poderá ver a tabela de todos os vectores possíveis de serem usados no Spectrum, com a indicação do endereço inicial da rotina de interrupt.

Esta instrução não afecta registos ou flags.

Já vimos duas instruções usadas para colocar os vectores em I: LD I,A e LD A,I (esta última para identificar o vector).

As instruções RST n têm funções atribuídas, pois encontram-se todas na ROM. Vê-las-emos quando falarmos das rotinas úteis da ROM para utilização directa. Para já basta saber que a função que executam é similar a uma instrução CALL.

As instruções que se seguem são específicas de interrupts.

### TIPO: DI

**FUNÇÃO:** Esta instrução pode ser lida como "desarma interrupts", do inglês Disable Interrupts. A sua execução faz com que o Z80 ignore interrupts e manter-se-á assim até serem "armados" de novo. Não afecta flags ou registos.

### TIPO: EI

**FUNÇÃO:** Esta instrução pode ser lida como "arma interrupts" (em inglês, Enable Interrupts). A sua execução permite o reconhecimento posterior de interrupts, MAS SÓ DEPOIS DE MAIS UMA INSTRUÇÃO TER SIDO EXECUTADA! Por isto, muitas rotinas terminam com

.....  
.....  
EI  
RET

Isto sucede porque os interrupts são processados em série. Esta instrução não afecta registos ou flags.

### TIPO: RETI

**FUNÇÃO:** Esta instrução significa "retorne do interrupt". É exactamente igual a RET, mas avisa também o sistema de I / O (no

caso a ULA) do fim da rotina de interrupt. Não afecta flags ou registos.

#### TIPO:RETN

**FUNÇÃO:** Esta instrução significa “retome do interrupt não desarmável”. É uma instrução sem interesse para o Spectrum, uma vez que não podemos usar este tipo de interrupt. Não afecta registos ou flags.

Veremos, nos exemplos de programas, uma rotina que usa interrupts.

O próximo capítulo tratará de notação BCD.

## CAPÍTULO 17

### NOTAÇÃO BCD

Basicamente, a notação BCD leva mais tempo e gasta mais memória, constituindo uma esplêndida fonte de erros.

Esta notação é uma forma de representar informação em formato decimal, como foi dito atrás. Para representarmos cada um dos dígitos de 0 a 9, necessitamos de apenas 4 bit e não são usados seis dos códigos possíveis nesta representação.

Isto quer dizer que cada byte pode substituir ou codificar dois dígitos — notação BCD. Vejamos os “nibble” (conjunto de 4 bit) correspondentes a cada um dos dígitos.

NIBBLE	ALGARISMO	NIBBLE	ALGARISMO
0000	0	0101	5
0001	1	0110	6
0010	2	0111	7
0011	3	1000	8
0100	4	1001	9

Os "nibble", não usados são 1010, 1011, 1100, 1101, 1110 e 1111. Esta convenção nada tem a ver com o tipo de "raciocínio" de computador, pois, por exemplo a soma

```
BCD 06 0000 0110
BCD 09 0000 1001
-----
BCD 15 0000 1111
```

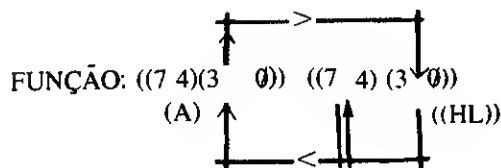
não dá um número BCD válido. Para que o resultado seja um número BCD legítimo, temos a instrução DAA, do inglês Decimal Adjust Acumulador, que soma 6 se o resultado for maior que 9, para uma adição ou tira 6 se o resultado for menor que 0, para a subtração. Esta instrução assume que o conteúdo do Acumulador é a soma ou a diferença de operandos BCD. Esta instrução afecta ainda todas as flags menos a N, cuja única razão de existência é precisamente avisar a instrução DAA se é soma ou subtração de operandos que está a ser realizada. A flag P/O funciona como flag de paridade. Esta instrução só pode ser usada após instruções que obedecem às seguintes condições:

- 1 — coloquem o seu resultado no Acumulador;
- 2 — afectam devidamente as flags C, AC e N.

Isto implica que DAA não pode ser usada após INC, DEC ou quaisquer instruções de 16 bit que coloquem o seu resultado nos registos IX, IY ou no par HL.

Neste grupo incluímos ainda mais duas instruções que fazem rotação de números BCD.

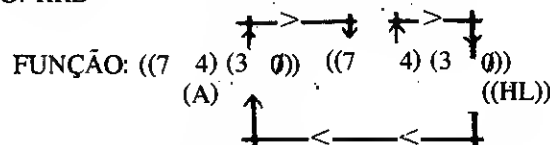
TIPO: RLD



O primeiro byte representa o conteúdo do Acumulador e o segundo o byte endereçado por HL.

FLAGS: C- NA Z- X S- X P / O- P AC- 0 N- 0

TIPO: RRD



O primeiro byte representa o conteúdo do Acumulador e o segundo o byte endereçado por HL.

FLAGS: C- NA Z- X S- X P / O- P AC- 0 N- 0

Como se pode ver, a notação BCD é uma grande trapalhada, o que torna extremamente rara. Fica aqui explicada para os curiosos...

O nosso último capítulo sobre as instruções do Z80 tratará de todas as que ainda não foram comentadas.

## CAPÍTULO 18

### O RESTO

#### INSTRUÇÕES DE STATUS

Não, não é status social, mas o estado da Carry flag. Temos duas instruções que nos permitem alterar directamente o seu estado e que são:

TIPO: SCF (Set Carry Flag)

FUNÇÃO: Esta instrução faz Carry= 1, AC= 0 e N= 0.

TIPO: CCF (Complement Carry Flag)

FUNÇÃO: Se Carry= 1 faz Carry= 0; se Carry= 0 faz Carry= 1 — operação complemento. Afecta ainda a flag AC de modo desconhecido e faz N= 0.

#### INSTRUÇÕES SEM DESIGNAÇÃO ESPECIAL

TIPO: NOP (No Operation)

FUNÇÃO: Nada faz, para além de refrescar as memórias voláteis e adicionar 4 ciclos de relógio ao programa, que é o seu tempo de execução.

TIPO: HALT

FUNÇÃO: O Z80 suspende a sua actividade, realizando NOPs. Necessita de um interrupt ou de um reset para que o Z80 reassuma as suas funções. Instrução muito pouco útil. O seu uso no Spectrum leva geralmente a “crash”.

#### COMPLEMENTANDO O ACUMULADOR

TIPO: CPL

FUNÇÃO: Complementa o Acumulador (“one’s complement”). Suponha que o A= 10110101 B. Após a execução de CPL, A= 01001010 B (o que estava set fica reset e vice-versa).

FLAGS: AC- 1 N- 1 As outras não são afectadas.

TIPO: NEG

FUNÇÃO: Nega o Acumulador (“two’s complement”). É equivalente a subtrair o conteúdo do Acumulador de zero. Note que # 80 não é afectado.

FLAGS: C- X Z- X S- X P/ O- O AC- X N- 1

E assim vimos todas as instruções do Z80. A segunda parte deste livro referirá aplicações práticas de Assembler e particularidades no uso do Spectrum.

Penso que deste modo, todos poderão beneficiar deles. E agora, se necessita do HEX LOADER, ligue o seu computador, introduza o programa, faça RUN 3010 e guarde a cópia.que gravou.

## HEX LOADER

### CAPÍTULO 19

### HEX LOADER

Uma vez que, nesta segunda parte, a prática é fundamental para uma melhor compreensão e pensando naqueles leitores que, por perguça ou falta de disponibilidades pecuniárias, não possuem ainda um Assemblador e / ou um Monitor, apresento aqui um programa em BASIC muito simples e cuja finalidade, é permitir introduzir códigos hexadecimais na memória do Spectrum, obter a sua listagem no écran ou na impressora, fazer SAVE e LOAD.

O programa HEX LOADER tem a simplicidade do BASIC e, como tal, não me deterei a explicá-lo, pois presumo que o leitor interessado tem uma proficiência razoável nesta linguagem. De qualquer modo, apresenta menu de opções e foi estruturado de modo a ser à prova de "asneiras". Poderá construir programas em máquina a partir do endereço 27500 decimal até ao limite da memória.

Para que seja útil, os exemplos apresentados terão:

- 1 — Endereço em HEX;
- 2 — Código (s) HEX da linha de Assembler;
- 3 — Mnemónica;
- 4 — Comentários, quando apropriado.

```

1 REM * * * * *
2 REM
3 REM   HEX LOADER
4 REM
5 REM © J. PAULO FRAGOSO 1984
9 REM
10 REM * * * * *
11 GO TO 25
20 CLEAR 27499
25 POKE 23609, 120: POKE 23658, 8
26 LET O= PI- PI: LET P= 1: LET G= 500: LET
TS= "0123456789ABCDEF"
30 CLS: PRINT AT P, 12: "MENU"
40 PRINT AT 3, 0: "1 — INPUT CODIGO HEX"
50 PRINT AT 5, 0: "2 — SAVE CODIGO HEX"
60 PRINT AT 7, 0: "3 — LIST CODIGO HEX"
70 PRINT AT 9, 0: "4 — LOAD CODIGO HEX"
75 PRINT AT 11, 0: "5 — RUN CODIGO HEX"
80 LET I= 0: LET I= CODE INKEY$: IF I < 49
OR I > 53 THEN GO TO 80
90 GO TO (G * (I= 49)) + (2 * G * (I= 50)) +
(3 * G * (I= 51)) + (4 * G * (I= 52)) + 5 * G *
(I= 53))
499 REM INPUT
500 CLS: INPUT "ENDEREÇO INICIAL?": S: IF
S < 27500 THEN PRINT "TERÁ DE SER ACIMA
DE 27500": PAUSE 150: GO TO G
505 LET START= S: CLS: PRINT AT 7, 0: "PARA
VOLTAR AO MENU PRIMA Z"
506 LET H$= ""
510 IF H$= "" THEN INPUT "HEX": H$

```



```

520 IF HS(P)= "Z" THEN GO TO 30
530 IF LEN HS / 2 < > INT (LEN HS / 2) THEN
PRINT "ERRO NO INPUT": GO TO 506
540 LET J= 0
550 FOR L= 16 TO P STEP -15
560 LET H= CODE HS((L= 16) + 2 * (L= P))
570 IF H < 48 OR H > 70 OR (H > 57 AND H < 65)
THEN PRINT "ERRO NO INPUT": GO TO 506
580 LET J= J + L * ((H < 58) * (H-48) + (H > 64
AND H < 71) * (H-55))
590 NEXT L: POKE S,J: LET S= S + 1: PRINT HS
(TO P + P): " ": LET HS= HS(3 TO): GO TO 510
999 REM SAVE
1000 CLS: INPUT "NOME DO PROGRAMA?": NS: IF
LEN NS > 10 THEN PRINT "NOME MUITO
GRANDE": PAUSE 150: PAUSE 150: GO TO G + G
1010 PRINT AT 5,0: "VERIFIQUE AS LIGAÇÕES AO
GRAVADOR": SAVE NSCODE START, S + P -
START
1020 PRINT AT 7,0: "DESEJA FAZER VERIFY? (S / N)"
1030 LET I= 0: LET I= CODE INKEY$: IF I < > 83
AND I < > 78 THEN GO TO 1030
1040 IF I= 83 THEN CLS: PRINT AT 7,0: "VERIFIQUE
AS LIGAÇÕES AO GRAVADOR PARA VERIFI-
CAR": NS: VERIFY NSCODE
1050 PRINT AT 12,0: "VERIFICADO": AT 14,0: "PRI
MA QUALQUER TECLA PARA MENU": PAUSE
0: GO TO 30
1499 REM LIST
1500 CLS: PRINT AT 7,0: "DESEJA IMPRIMIR OU VER?"
1510 LET I= 0: LET I= CODE INKEY$: IF I < > 73
AND I < > 86 THEN GO TO 1510
1520 IF I= 73 THEN GO TO 1750
1530 FOR N= START TO S STEP 5
1540 FOR M= 0 TO 4
1550 LET Y= INT (PEEK (N + M) / 16): LET X= PEEK

```

```

(N + M) — 16 * INT (PEEK (N + M) / 16)
1560 PRINT TAB 6 + M * 5; TS(Y + P); TS(X + P);
1570 NEXT M
1580 NEXT N
1590 PAUSE 0: PAUSE 0: GO TO 30
1750 FOR N= START TO S STEP 5
1760 FOR M= 0 TO 4
1770 LET Y= INT (PEEK (N + M) / 16): LET X= PEEK
(N + M) -16 * INT (PEEK (N + M) / 16)
1780 LPRINT TAB 6 + M * 5; TS(Y + P); TS(X + P);
1790 NEXT M
1800 NEXT N
1810 GO TO 30
1999 REM LOAD
2000 CLS: PRINT AT 1,0: "VERIFIQUE AS LIGAÇÕES
AO GRAVADOR. NOTE QUE SE O PROGRAMA
COMEÇAR ANTES DE 27500, ESTE PROGRAMA
COLOCA-LO-A AUTOMATICAMENTE A PARTIR
DE 27500"
2010 LOAD ""CODE 27500
2020 GO TO 30
2499 REM RUN
2500 CLS: PRINT AT 1,0, "FEZ UMA COPIA EM CAS
SETTE DO PROGRAMA DE CODIGO"— MAQUI
NA? SE O DESEJA FAZER PRIMA M PARA VOL
TAR AO MENU; QUALQUER OUTRA TECLA PA
RA CONTINUAR."
2510 LET I= 0: CODE INKEY$: IF I= 0 THEN GO TO 2510
2520 IF I= 77 THEN GO TO 30
2530 INPUT "ENDEREÇO INICIAL?": W: CLS: PRINT
USR W: PAUSE 0: GO TO 30
3000 REM SAVE HEX LOADER
3010 SAVE "HEX LOADER" LINE 20: RUN 20

```

Versão 1: usar (HL) e A.

END. HEX	ASSEMBLER
	ORG # 6D00
6D00 21006C	LD HL,# 6C00 ; inicialização
6D03 7E	LD A, (HL) ; primeiro número
6D04 23	INC HL
6D05 86	ADD A, (HL) ; é somando ao segundo
6D06 23	INC HL
6D07 77	LD (HL),A ; e finalmente guardado
6D08 C9	RET ; retorno ao BASIC

Depois de introduzir o programa na memória, há que dar valores aos byte indicados (# 6C00 e # 6C01). Poderá para tal usar POKE, a partir do BASIC (não se esqueça de CLEAR 27647, caso não esteja a utilizar o HEX LOADER e isto ANTES de meter o código ou a assemblagem). Para obter o resultado faça PEEK. Poderá misturar com BASIC:

```
10 INPUT A,B
20 POKE 27648,A: POKE 27649,B
30 RANDOMIZE USR 27904
40 PRINT PEEK 27650
50 GO TO 10
```

Fartar-se-á logo que introduza dois números cuja soma seja superior a 255, pois começarão a aparecer erros nestes casos. Antes de resolvermos tal problema vejamos outra versão, que usa endereçamento directo.

Versão 2:

	ORG # 6D00
6D00 3A006C	LD A (# 6C00) ; primeiro número
6D03 47	LD B,A
6D04 3A016C	LD A, (# 6C01) : segundo número
6D07 86	ADD A,B
6D08 32026C	LD (# 6C02),A ; o resultado é guardado
6D0B C9	RET

## CAPÍTULO 20

### APRENDENDO A SOMAR

Neste capítulo veremos alguns pequenos programas em Assembler que constituem os nossos primeiros passos neste tipo de programação. Antes de começarmos, importa esclarecer que, de um modo geral, os programas assumem que os byte de memória usados para guardar resultados contêm inicialmente # 00, salvo quando mencionado o contrário. Todos os programas apresentados neste capítulo iniciam-se em # 6D00 (ou 27904 decimal) e os byte a partir de # 6C00 (27648) até # 6CFF servirão para guardarmos valores ("data" ou dados). Os programas apresentam ainda diversos modos de abordar o problema proposto (não há dois programadores que façam um programa da mesma maneira para um mesmo fim, geralmente), visando esclarecer e não propriamente demonstrar a maneira mais eficiente, em termos de poupança de memória e /ou tempo.

**PROBLEMA:** somar dois números de 8 bit (partindo do princípio de que a sua soma é inferior ou igual a # FF), encontrando-se o primeiro em # 6C00 e o segundo em # 6C01. Colocar o resultado em # 6C02.

Esta versão gasta mais byte, por causa do endereçamento directo. Gasta 57 ciclos de relógio, ao passo que a primeira gasta apenas 37! Moral da história: sempre que fizer operações sobre byte consecutivos de memória, use endereçamento indirecto. Note, porém, que necessitará de inicializar HL.

Vejamos agora o mesmo problema re-equacionado.

**PROBLEMA:** somar dois números de 8 bit, encontrando-se o primeiro em # 6C00 e o segundo em # 6C01. Garantir que o resultado seja correcto, caso exceda # FF. Colocar o resultado em # 6C03 e # 01 em # 6C02, se a soma exceder # FF. Iniciar o programa em # 6D00.

Versão 1:

```

6D00 21006C    LD HL,# 6000 ; inicializar HL
6D03 7E        LD A,(HL)   ; primeiro número
6D04 23        INC HL
6D05 A7        AND A       ; garantir que a Carry=0
6D06 86        ADD A,(HL)  ; efectuar soma
6D07 23        INC HL
6D08 3002      JR NC,FIM    ; salta para #6D0C se
                          Carry= 0
6D0A CBCE      SET 0,(HL)   ; (# 6C02)= # 01
6D0C 23        FIM INC HL
6D0D 77        LD (HL),A    ; resultado guardado em
                          # 6C03
6D0E C9        RET

```

A única linha do BASIC que temos de alterar é:

40 PRINT (PEEK 27650 + 256 \* PEEK 27651): POKE 27650,0

A justificação da última instrução é simples: permitirmos que o byte # 6C02 seja reinicializado para cálculos posteriores. Vemos que agora podemos calcular qualquer soma de números de 8 bit.

Versão 2: O mesmo problema, mas guardando o resultado no par BC.

Note que # 6C02 — # 6C03 guardando o conteúdo de BC.

```

6D00 010000    LD BC,# 0000 ; inicializar BC
6D03 21006C    LD HL,# 6C00 ; inicializar HL
6D06 7E        LD A,(HL)    ; primeiro número
6D07 23        INC HL
6D08 A7        AND A        ; garantir que Carry= 0
6D09 86        ADD A,(HL)   ; efectuar soma
6D0A 23        INC HL
6D0B 4F        LD C, A      ; guardar LSB em C
6D0C 3002      JR NC, FIM   ; salta para# 6D10 se Carry= 0
6D0E 0601      LD B,# 01    ; se Carry= 1
6D10 71        FIM LD (HL), C ; colocar LSB em# 6C02
6D11 23        INC HL
6D12 70        LD (HL), B   ; colocar MSB em# 6C03
6D13 C9        RET

```

De novo, temos alteração na linha 40:

40 PRINT (PEEK 27650 + 256 \* PEEK 27651)

Repare que, neste caso, o programa em código restaura sempre os valores dos byte usados para guardar o resultado. Note também que, se correr este programa a partir do HEX LOADER (após colocar os valores a somar nos byte apropriados através de um comando directo e retornar ao HEX LOADER com GO TO 11 ou GO TO 25, para não destruir as variáveis), obterá o valor correcto imprimido no écran. Isto deve-se ao facto do HEX LOADER imprimir no écran o conteúdo do par BC ao regressar ao BASIC, uma vez que o comando que usa é PRINT USR endereço. A função USR seguida de um número (o seu argumento) dá como resultado o conteúdo do par BC (pág. 180 do Manual do Spectrum).

**PROBLEMA:** Somar dois números de 16 bit, estando o primeiro em # 6000 - # 6C01 e o segundo em # 6C02 - # 6C03. Co-

locar o resultado em # 6C04 - # 6C05. Começar o programa em # 6D00. Não usar instruções de soma de 16 bit; usar apenas instruções de soma de 8 bit (soma em cadeia).

Versão 1: endereçamento indirecto por HL.

		ORG # 6D00	
6D00	21006C	LD HL, # 6C00	; inicialização de HL
6D03	7E	LD A, (HL)	; LSB do primeiro número
6D04	23	INC HL	
6D05	23	INC HL	
6D06	A7	AND A	; garantir Carry = 0
6D07	86	ADD A, (HL)	; somar LSB's dos números
6D08	23	INC HL	
6D09	23	INC HL	
6D0A	77	LD (HL), A	; guardar LSB do resultado
6D0B	2B	DEC HL	
6D0C	7E	LD A, (HL)	; MSB do segundo número
6D0D	2B	DEC HL	
6D0E	2B	DEC HL	
6D0F	8E	ADC A, (HL)	; somar MSB's dos números
6D10	23	INC HL	
6D11	23	INC HL	
6D12	23	INC HL	
6D13	23	INC HL	
6D14	77	LD (HL), A	; guardar MSB do resultado
6D15	C9	RET	; retorno ao BASIC

Como exercício poderá construir as linhas de BASIC para dar valores aos byte de data16 e para ler o resultado. Note que a rotina actualiza os byte onde coloca o resultado, pois ao colocar os novos valores apaga os anteriores. Verá que o programa erra para resultados superiores a # FFFF.

Versão 2: usando endereçamento indexado:

		ORG # 6D00	
6D00	DD21006C	LD IX, # 6C00	; inicializando IX
6D04	DD7E00	LD A, (IX + 0)	; LSB do primeiro número
6D07	A7	AND A	; garantir que Carry = 0
6D08	DD8602	ADD A, (IX + 2)	; somar LSB's dos números
6D0B	DD7704	LD (IX + 4), A	; guardar LSB do resultado
6D0E	DD7E01	LD A, (IX + 1)	; MSB do primeiro número
6D11	DD8E03	ADC A, (IX + 3)	; somar MSB's dos números
6D14	DD7705	LD (IX + 5), A	; guardar MSB do resultado
6D17	C9	RET	; retomo ao BASIC

Repare que esta versão não só usa mais byte como demora mais tempo a ser executada. No entanto, demonstra o uso dos registos de indexamento. Também erra se o resultado for superior a # FFFF.

PROBLEMA: somar dois números de 16 bit, estando o primeiro em # 6C00 - # 6C01 e o segundo em # 6C02 - # 6C03. Colocar o resultado em # 6C04 - # 6C05. Começar o programa em # 6D00. Use instruções de soma de 16 bit.

		ORG # 6D00	
6D00	2A006C	LD HL, (# 6C00)	; recolher primeiro número
6D03	ED5B026C	LD DE, (# 6C02)	; recolher segundo número
6D07	19	ADD HL, DE	; efectuar soma
6D08	22046C	LD (# 6C04), HL	; guardar resultado
6D0B	C9	RET	; retomo ao BASIC

Realmente bem mais prático, mas continua a dar erro para resultados superiores a # FFFF.

PROBLEMA: considere o problema acima, mas garantindo que o resultado final é correcto, se exceder # FFFF; neste caso coloque # 01 em # 6C06 (ou # 00 se não passar de # FFFF).

		ORG # 6D00	
6D00	2A006C	LD HL, (#6C00)	; recolher primeiro número
6D03	ED5B026C	LD DE, (# 6C02)	; recolher segundo número
6D07	AF	XOR A	; Acumulador= # 00
6D08	A7	AND A	; Carry= 0
6D09	19	ADD HL,DE	; efectuar soma
6D0A	22046C	LD (# 6C04),HL	; guardar resultado
6D0D	3001	JR NC, FIM	; se Carry= 0 vai para # 6D10
6D0F	3C	INC A	; Acumulador= # 01
6D10	32066C	LD (# 6C06),A	; guardar eventual carry
6D11	C9	RET	; retorno ao BASIC

Note que saberemos o resultado fazendo PRINT PEEK 27654: se obtivermos zero o resultado ocupa apenas 2 byte (é inferior ou igual a # FFFF); se obtivermos 1, o resultado será lido correctamente com o comando:

PRINT (PEEK 27652 + 256 \* PEEK 27653) + 65536

Finalizarei este capítulo dedicado às somas com alguns exercícios para o leitor praticar.

Exercícios: resolva os mesmos problemas, mas para a operação subtracção. Note que o método de extrair um resultado negativo correcto de uma subtracção é análogo ao da soma: teste a Carry flag. E lembre-se de que só há instruções de subtracção de 16 bit com a Carry flag (SBC). É a programar que aprenderá a dominar Assembler!

## CAPÍTULO 21

### ALGUMAS ROTINAS ÚTEIS DA ROM

Muitas das rotinas que necessitamos já se encontram na ROM do seu Spectrum. Para as usarmos temos, no entanto, de saber onde estão e os parâmetros que necessitam.

Veremos em primeiro lugar aquelas que podemos aceder com apenas um byte, através da instrução RST # NN.

RST # 00 — É equivalente a desligar e ligar o computador.

Faz o reset do sistema, continuando através de um salto absoluto em # 11CB (NEW).

RST # 08 — Esta rotina controla as mensagens de erro. O stack é limpo e a mensagem produzida.

RST # 10 — A rotina de imprimir o conteúdo do Acumulador. Prossegue em # 15F2. Veremos como usar.

RST # 18 — A rotina que vai buscar o carácter.

RST # 20 — A rotina que vai buscar o carácter seguinte.

RST # 28 — A rotina de entrada na “calculadora” de vírgula flutuante. Salta para # 335B.

RST # 30 — A rotina de criação de espaço no “work space” (espaço de trabalho).

RST # 38 — A rotina dos interrupts desarmáveis. Actualiza o relógio e lê o teclado (faz CALL #02BF).

Vejamos como usar RST # 10, que imprime caracteres válidos no écran.

## IMPRIMINDO NO ÉCRAN

Para obviar o trabalho de manipular directamente o écran, que será tema do próximo capítulo, podemos usar duas rotinas da ROM para o fazer. Qualquer delas terá de ser precedida pela sequência seguinte, que abre o canal S (de Screen).

HEX	ASSEMBLER
3E02	LD A,# 02
CD0116	CALL #1601

Em seguida, teremos instruções do tipo LD A,#NN, seguidas por RST # 10, até finalizarmos com RET. RST # 10 permite não só imprimir qualquer carácter no écran, como também mudar a posição presente de impressão, AT, TAB, comandos de uma tecla e cores temporárias. O seguinte programa ilustra estes aspectos (usa ainda uma rotina que explicaremos adiante).

HEX	ASSEMBLER	COMENT.
3E02	LD A,# 02	; ABRIR CANAL “S”
CD0116	CALL # 1601	
0618	LD B,# 18	; LIMPAR 24 LINHAS DO ECRAN
CD440E	CALL # 0E44	
3E16	LD A,# 16	; AT 10,8

D7	RST #10	
3E0A	LD A,# 0A	
D7	RST # 10	
3E08	LD A,# 08	
D7	RST # 10	
3E10	LD A,#10	; INK 2
D7	RST # 10	
3E02	LD A,#02	
D7	RST # 10	
3E11	LD A,# 11	; PAPER 6
D7	RST # 10	
3E06	LD A,# 06	
D7	RST # 10	
3E5A	LD A,# 5A	; PRINT “Z”
D7	RST # 10	
3E38	LD A,# 38	; PRINT “8”
D7	RST # 10	
3E30	LD A,# 30	; PRINT “0”
D7	RST # 10	
C9	RET	

Pode colocar este programa onde quiser, acima de RAMTOP. Há outro método de fazer algo semelhante: impressão no écran de strings. Para tal, coloca-se em DE o endereço inicial e o seu comprimento em BC. Depois é só chamar a rotina em # 203C:

END.	HEX	ASSEMBLER	COMENT.
		ORG# 6D00	
6D00	3E02	LD A,# 02	; ABRIR CANAL “S”
6D02	CD0116	CALL# 1601	
6D05	0618	LD B,# 18	; LIMPAR 24 LINHAS
6D07	CD440E	CALL# 0E44	
6D0A	11146D	LD DE,# 6D14	; INÍCIO
6D0D	011200	LD BC,# 0012	; COMPRIMENTO
6D10	CD3C20	CALL# 203C	; IMPRIME
6D13	C9	RET	; RETORNA AO BASIC
6D14	11001005	DEFB # 11, #00, # 10, #0 5	
6D18	160A085A	DEFB # 16, # 0A, # 08	
6D1C	38302041	DEFM “Z80 ASSEMBLER”	
6D20	5353454D		
6D24	424C4582		

Isto equivale ao BASIC:

```
CLS: PRINT PAPER 0; INK 5; AT 10 8; "Z80 ASSEMBLER"
```

Vejamos de seguida como produzir som através de código de máquina. Fundamentalmente, podemos usar duas rotinas da ROM. A primeira é conhecida por BEEPER e permite-nos criar sons mais versáteis do que a segunda, que é a rotina de comando BEEP.

A subrotina BEEPER controla o altifalante do Spectrum. Este é activado com o bit D4 igual a zero durante uma instrução OUT que use a porta 254 (# FE). De modo similar, quando D4 é igual a um, o altifalante é desactivado. O som é produzido na base de sequências muito rápidas liga-desliga.

Para usarmos esta subrotina teremos de colocar em DE a frequência da som multiplicada pela duração ( $f * t$ ) e em HL o valor resultante da equação  $HL = 437500 / f - 30.125$ , após o que chamamos a subrotina com CALL # 03B5.

A rotina BEEP é mais complicada de usar, pois recorre à calculadora. Essencialmente, entra-se nesta rotina com dois números no stack da calculadora: o do topo é a "altura" da nota e o imediatamente abaixo é a sua duração. Depois chama-se a rotina BEEP com CALL # 03F8. Chamo aqui a atenção do leitor para a relativa dificuldade de manipular a calculadora do Spectrum. Teremos oportunidade de a ver com algum detalhe mais adiante.

Recorrendo ao código de máquina podemos fazer SAVE, LOAD, MERGE e VERIFY de programas. Em particular é-nos permitido usar a ROM para gravarmos e lermos programas sem "header" (o primeiro bloco que é carregado). Veremos de seguida a rotina SA-BYTES e LD-BYTES. Em qualquer dos casos o par DE deverá conter o comprimento do bloco de bytes a ser tratado, o registo IX o endereço inicial do bloco e o Acumulador o valor # 00 para um header, ou valor # FF para um bloco de programa ou data. Depois basta fazer CALL # 04C2 para SAVE ou CALL # 0556 para LOAD. Adicionalmente, quando fazendo LOAD, a Carry deve ser igual a 1.

Exemplo de SAVE e LOAD de um bloco de programa sem header:

SAVE

```
LD A,# FF
LD IX,# NNNN ; INÍCIO
LD DE,# NNNN ; COMPRIMENTO
CALL# 04C2 ; SAVE
RET
```

LOAD

```
LD A,# FF
LD IX,# NNNN ; INÍCIO
LD DE,# NNNN ; COMPRIMENTO
SCF          ; CARRY= 1
CALL# 0556   ; LOAD
RET
```

A subrotina que se segue é conhecida por CLEAR-LINES ou CL-LINES. A sua finalidade é apagar x linhas do écran a contar da linha de baixo. Antes de a chamar há que carregar o registo B com o número de linhas a apagar. Depois, CALL # 0E44.

Exemplo: apagar as 16 linhas inferiores (partindo do princípio de que o canal "S" está aberto).

```
LD B,# 10      ; APAGAR AS 16 LINHAS INFERIORES
CALL# 0E44
RET
```

Chamo aqui a atenção do leitor para o pormenor de assegurar a abertura do canal "S". Abaixo veremos como abrir os diversos canais do Spectrum, embora já tenhamos utilizado atrás essa rotina. Note que tanto esta rotina como a seguinte não afectam o canal em uso.

Vamos agora ver a rotina CL-SCROLL, que permite o scroll de x linhas do écran, terminando com o recurso a CL-LINES para limpar a linha inferior. O número de linhas a fazer scroll é colocado em B, mas obedecendo à seguinte expressão:  $B = \text{número real de linhas a serem afectadas} - 1$ . Exemplo: fazer scroll a todas as linhas menos as duas de cima (assumindo o canal "S" aberto).

```
LD B,# 15      ; SCROLL DE TODO O ECRAN, MENOS AS
CALL# 0E00     ; DUAS LINHAS SUPERIORES
RET
```

A unidade standard do Spectrum, isto é sem o Interface 1 ligado, permite que o utilizador tenha acesso a três deles sem aparecer a reportagem de erro "INVALID STREAM": são os canais "K", "S" e "P". Os canais #00 e #01 são iguais, abrindo o canal "K" - a(s) linha(s) inferior(s) do écran, onde aparecem as reportagens do sistema operativo, inputs, etc. O canal #02 é o canal "S", o resto do écran. Finalmente o canal #03 é o canal "P", da impressora (ZX PRINTER ou TS 2040). Os canais são abertos colocando no Acumulador o valor apropriado e chamando a rotina em #1601. Exemplo: abrir o canal "P"

```
LD A, #03 ; SINALIZAR IMPRESSORA
CALL# 1601 ; ABRIR CANAL "P"
RET
```

Para terminar esta breve passagem pela ROM, mencionaremos agora como ler o teclado do Spectrum a partir da linguagem máquina.

Podemos fazê-lo de várias maneiras:

- 1 — leitura indirecta através da variável do sistema LAST-K;
- 2 — usando instruções IN como a rotina KEY-SCAN em #028E;
- 3 — chamar a rotina KEY-SCAN e comparar os valores obtidos com valores conhecidos;
- 4 — reproduzir a rotina de INKEY\$ em #2646.

Deixo o leitor com uma rotina de input alfa-numérico que recorre ao método 4 e com a recomendação de estudar o excelente livro "THE COMPLETE SPECTRUM ROM DISASSEMBLY", onde poderá acompanhar as rotinas que aqui vimos (e todas as outras).

A rotina de input que abaixo se segue obedece ao formato das rotinas apresentadas no Capítulo 20. O leitor tem a minha permissão para a usar nos seus programas **particulares**. Qualquer uso desta rotina em programas comerciais sem a minha autorização escrita é ilegal e constitui violação dos direitos de autor.

6D00	21FE6C	INÍCIO	ORG # 6D00 LD HL, # 6CFE	; indicador do último caract. ; ter entrado
6D03	3600		LD (HL), # 00	
6D05	23		INC HL	
6D06	366C		LD (HL), # 6C	
6D08	CD8E02	KEYSCN	CALL # 028E	; busca do teclado
6D0B	0E00		LD C, # 00	
6D0D	20F9		JR NZ, KEYSCN	; nada premido ou demais
6D0F	CD1E03		CALL # 031E	; teste valor da tecla
6D12	30F4		JR NC, KEYSCN	; valor ilegal
6D14	15		DEC D	; preparação de valores para
6D15	5F		LD E, A	; a rotina que se segue
6D16	CD3303		CALL # 0333	; descodificação do valor da
				; tecla premida
6D19	FE0D		CP #0D	; é ENTER?
6D1B	381C		JR C, BUZZER	; não, é ilegal
6D1D	2846		JR Z, BIPBIP	; sim, prossegue adiante
6D1F	FE20		CP # 20	; é SPACE?
6D21	3816		JR C, BUZZER	
6D23	2840		JR Z, BIPBIP	
6D25	FE30		CP # 30	; são números?
6D27	3810		JR C, BUZZER	
6D29	FE3A		CP # 3A	
6D2B	3838		JR C, BIPBIP	
6D2D	FE41		CP # 41	; são letras?
6D2F	3808		JR C, BUZZER	
6D31	FE5B		CP # 5B	
6D33	3830		JR C, BIPBIP	; é SYMBOL SHIFT + 0?
6D35	FE5F		CP # 5F	; sim. Vai para DELETE
6D37	280B		JR Z, DELETE	; valores ilegais fazem soar
6D39	112800	BUZZER	LD DE, # 0028	; um sinal grave
6D3C	215211		LD HL, # 1152	; BEEPER
6D3F	CDB503		CALL # 03B5	; volta para nova tecla
6D42	18C4		JR KEYSCN	; último caract. entrado
6D44	2AFE6C	DELETE	LD HL, (# 6CFE)	; início do buffer de input
6D47	11006C		LD DE #6C00	; Carry= 0
6D4A	A7		AND A	; DELETE para trás do buffer
6D4B	ED52		SBC HL, DE	; não é permitido
6D4D	28EA		JR Z, BUZZER	; restaura HL
6D4F	2AFE6C		LD HL, (# 6CFE)	; efectua DELETE
6D52	2B		DEC HL	; no buffer
6D53	22FE6C		LD (#6CFE), HL	; e na var. do écran
6D56	2A845C		LD HL, (# 5C84)	
6D59	2B		DEC HL	
6D5A	22845C		LD (# 5C84), HL	; efectua DELETE no écran
6D5D	11003D		LD DE, # 3D00	; PRINT ALL-CHARCTS.
6D60	CD7F0B		CALL # 0B7F	; executa
6D63	181A		JR BIIIIP	; recolhe última posição e
6D65	2AFE6C	BIPBIP	LD HL, (# 6CFE)	



6D68	23	INC HL	; incrementa-a antes de de-
6D69	22FE6C	LD (# 6CFE),HL	; volver indicador
6D6C	77	LD(HL),A	; coloca carácter no buffer
6D4D	F5	PUSH AF	; preserva carácter
6D6E	3E02	LD A, # 02	; canal "S"
6D70	CD0116	CALL # 1601	; CHAN-OPEN
6D73	F1	POP AF	; retoma carácter
6D74	D7	RST # 10	; e imprime-o
6D75	2AFE6C	LD HL, (# 6CFE)	; teste ao fim do buffer
6D78	F5	PUSH AF	; preserva carácter
6D79	3E3F	LD A, # 3F	
6D7B	BD	CP L	; fim do buffer?
6D7C	CA006D	JP Z, INICIO	; sim. Input longo
6D7F	11F000 BIIIIP	LD DE, # 00F0	; sinal agudo de aceite
6D82	210402	LD HL, # 0204	
6D85	CDB503	CALL # 03B5	; BEEPER
6D88	F1	POP AF	; restaura carácter
6D89	FE0D	CP # 0D	; último teste a ENTER
6D8B	DA916D	JP C, FIMINP	; terminou input
6D8E	C3086D	JP, KEYSCN	; novo carácter
6D91	C9 FIMINP	RET	; FIM DA ROTINA

Note que # 6CFE-# 6CFF actua como indicador de 16 bit do último carácter aceite, sendo restaurado em INICIO. O Leitor poderá complementar a rotina com uma outra que seja chamada antes desta e que tenha por fim limpar o buffer de 63 caracteres (mais # 0D), bem como fazer uma rotina de tratamento do input, que poderá começar em # 6D91, substituindo o RET. Poderá colocar a rotina noutro local da memória, desde que substitua os endereços absolutos usados. Poderá ainda modificar a rotina para o canal "K" ou para imprimir num ponto qualquer do écran.

E agora vamos tratar da manipulação directa do écran e dos atributos.

## CAPÍTULO 22

### MANIPULANDO O ÉCRAN

Vimos como usar a ROM para imprimir caracteres no écran. No entanto, se pensamos criar outro tipo de efeitos gráficos, como a movimentação de "sprites" ou mesmo UDG's em alta resolução e sem o efeito de "pisca-pisca", há necessidade de manipular directamente o écran.

Se consultar o seu Manual do Spectrum na página 165 encontrará o mapa da memória RAM. Começa em 16384 (#4000) com o "DISPLAY FILE", isto é, onde se colocam as coisas no écran e que termina em 22527 (# 57FF) inclusive. São 6144 (#1800) byte. Depois temos o "ATTRIBUTE FILE", ou seja, como as coisas aparecem no écran, começando em 22528 (# 5800) e terminando em 23295 (# 5AFF) inclusive. São 768 (# 0300) byte. Para já podemos constatar que a notação HEX dá-nos números fáceis de trabalhar. Veremos que não é por acaso.

Passarei a referir o "Display file" pela abreviatura DIF e o "Attribute file" pela abreviatura ATF.

## O DIF

Importa que compreendamos como está organizado o DIF. Para tal, introduza o seguinte programa em BASIC:

```
10 FOR N= 10 TO 80 STEP 10
15 INK (RND * 6) + 1
20 CIRCLE 127,87,N
30 NEXT N
40 SAVE "CIRCULO" SCREEN$: NEW
```

O seu computador desenhara oito circunferências concêntricas de cores aleatórias e preparará-se para gravar a imagem. Coloque uma cassette no seu gravador e proceda à operação. No fim o Spectrum limpará toda a memória usada. Prima:

LOAD ""SCREEN\$

Rebobine a fita e observe a entrada da imagem: corre a linha superior de pixels, depois salta sete, corre outra linha, salta sete e assim por diante até um terço do écran. Depois corre a segunda linha de pixels, salta sete, corre outra linha e vai repetindo este processo até toda a informação do primeiro terço estar no écran. O procedimento para o segundo e terceiro terços é similar. Seguem-se as cores: aqui o Spectrum vai introduzindo as ditas de acordo com as posições PRINT. Conclusões: o DIF é organizado em alta resolução e dividido em três blocos, com um arranjo que não parece fácil de manipular; o ATF está organizado em baixa resolução (a mesma do texto) e parece fácil de manipular.

Acontece que o DIF não é tão difícil de usar como parece, se nos concentrarmos na notação HEX. Vejamos os endereços do primeiro e último byte de cada bloco:

BLOCO	LINHAS	END. INICIAL	END. FINAL
1	0 — 7	# 4000	# 47FF
2	8 — 15	# 4800	# 4FFF
3	16 — 23	# 5000	# 57FF

Mais uma vez, os números HEX apontam para uma fácil manipulação do DIF. Vejamos agora o primeiro bloco em pormenor, (a discussão seguinte é análoga para os outros dois blocos).

Se nos lembramos que cada posição PRINT necessita de  $8 \times 8$  byte (64) e que há 32 destas posições em cada linha, o passo seguinte é desenvolvermos este raciocínio em notação HEX. 32 é # 20: um número simpático para trabalhar. Por outro lado, cada bloco contém  $32 \times 8 \times 8 = 2048$  byte, que em HEX dá # 0800: outro número simpático. Vimos que os byte correspondem à ordem: todos os primeiros byte do bloco, todos os segundos byte do bloco, etc., o que dá o salto de 7. Ora isto significa que, por exemplo, o primeiro byte da primeira posição PRINT do écran dista do segundo de  $32 \times 8 = 256$  ou # 0100: mais um número simpático!

Vamos tirar conclusões:

1 — para cada posição PRINT o primeiro byte dista dos seguintes um múltiplo de # 100;

2 — um byte de uma posição PRINT dista do seu byte análogo na posição PRINT imediatamente abaixo ou acima, num mesmo bloco, de # 20;

3 — um byte num bloco dista do byte com a mesma posição relativa noutro bloco qualquer de # 800 ou # 1600, consoante o bloco de referência e o bloco referenciado.

É altura de passarmos à prática e testarmos as nossas conclusões. Vamos construir um programa que nos preencha a preto a primeira posição PRINT do primeiro bloco:

6D00	210040	ORG # 6D00
6D03	3EFF	LD HL, # 4000 ; primeiro byte do DIF
6D05	110010	LD A, # FF ; todos os bit set (= 1)
6D08	0607	LD DE, # 0100 ; incrementador de endereço
6D0A	77	LD B, # 07 ; contador do loop
6D0B	19	LD (HL), A
6D0C	10FE	ADD HL, DE
6D0D	C9	DJNZ LOOP
		RET

Vamos agora verificar a segunda conclusão, preenchendo a preto todos os primeiros byte das posições PRINT do primeiro bloco. Para tal só necessitamos de alterar os byte # 6D06 - 6D07 do programa acima, dando:

```
6D05 110200 LD DE,# 0020
```

Até aqui tudo bem. Para testarmos a terceira conclusão vamos preencher a preto o primeiro byte de cada bloco. Podemos usar o mesmo programa, se alterarmos duas linhas para:

```
6D05 110008 LD DE,# 0800
6D08 0602 LD B, #02
```

Afinal, a manipulação do DIF não é a dificuldade em código de máquina que é em BASIC (já viu os POKE's que tinha de fazer e os números decimais que tinha de usar?), com a vantagem adicional de podermos usar as duas linhas inferiores do écran.

E quando queremos ter um UDG a passear de um bloco para outro, perguntará o leitor, como se resolve o caso limite? Nada mais simples! Repare que, por exemplo, o último byte do primeiro bloco, que corresponde ao byte de "baixo" da posição PRINT 15,31, tem o endereço # 4FFF. Ora os 32 byte seguintes correspondem aos primeiros byte das posições PRINT da linha 16. Isto significa que, nos casos limite, as nossas conclusões terão de ser ajustadas para:

4 — para cada posição PRINT da última linha do bloco 1 o primeiro byte dista do seu byte análogo na primeira linha do bloco 2 de #820 e de # 1620 se considerarmos o seu análogo no bloco 3.

5 — para cada posição PRINT da última linha do bloco 1 o último byte dista do primeiro byte da posição PRINT análoga na primeira linha do bloco 2 de # 20 e de # 820 se considerarmos a posição PRINT análoga ao bloco 3.

Isto quer dizer que, se o leitor está a pensar fazer o seu jogo "arcade" que irá ser o sucesso deste ano, terá de ter uma rotina para testar a posição dos UDG's (ou dos sprites, se REALMENTE o jogo pretender ser um SUCESSO) nos casos limite e substituir a rotina normal de movimentação dos mesmos, caso esta condição se verifique; sempre que tal hipótese não seja válida, o algoritmo referente às conclusões 1, 2 e 3 resolve o problema.

## O ATF

O ATF é muito mais simples de manipular, pois temos apenas 768 byte para nos preocupar e seguem todos a ordenação das posições PRINT. (Nota aos curiosos: podemos ter alta resolução de cor no Spectrum, criando um ATF equivalente ao DIF e em que cada byte do ATF original corresponde a oito no ATF expandido. Dadas as características da máquina e o desperdício de memória, nunca ninguém produziu uma rotina — que seria algo complicada — para criar este efeito.)

O algoritmo é evidente: cada byte afecta toda uma posição PRINT; logo, o byte análogo em coluna está a uma diferença do byte de referência de um múltiplo de #20.

O único cuidado a ter com estes byte é na sua construção. A página 116 do manual refere a função ATTR que nada mais é do que uma espécie de PEEK. A notação BIN é a melhor para estes byte, pois os bit 0,1 e 2 referem-se à cor da "tinta" (INK); os bit 3,4 e 5 ao "papal" (PAPER); o bit 6 ao "brilho" (BRIGHT) e o bit 7 ao "pisca-pisca" (FLASH). Estes três últimos têm o valor zero quando em "off" ou o valor um quando em "on". A tabela seguinte mostra quais os bit que controlam cada cor:

COR	INK		PAPER
BLACK (PRETO)	XXXXXX000	0	XXX000XXX
BLUE (AZUL)	XXXXXX001	1	XXX001XXX
RED (VERMELHO)	XXXXXX010	2	XXX010XXX
MAGENTA	XXXXXX011	3	XXX011XXX
GREEN (VERDE)	XXXXXX100	4	XXX100XXX
CYAN	XXXXXX101	5	XXX101XXX
YELLOW (AMARELO)	XXXXXX110	6	XXX110XXX
WHITE (BRANCO)	XXXXXX111	7	XXX111XXX

Simples, não é? Então vá em frente e construa algumas rotinas que lhe permitam fazer "scroll" em alta resolução nos quatro sentidos. Se lhe falta a coragem, acompanhe-me no próximo capítulo para ver como se protegem programas.

## CAPÍTULO 23

### PROTEÇÃO DE SOFTWARE

Finalmente terminou a sua obra prima! Naturalmente, está desejoso de a comercializar e ganhar dinheiro e fama. Mas como evitar que os piratas lhe roubem os proventos de meses de trabalho?

Gostaria de lhe dar um método simples, eficaz e cem por cento seguro, só que, infelizmente, tal método não existe e nenhum programa é inviolável, sendo isto particularmente sentido no Spectrum, devido à forma de implementação do sistema operativo. Assim, lembre-se que nenhum programa é inviolável, desde que o utilizador tenha acesso a código de máquina nesse computador.

O primeiro conselho que lhe dou é tirar uma listagem de boa qualidade do seu programa e garantir os seus direitos de autor (o conhecido "copyright") junto das entidades legais competentes. Registe a ideia, o nome do programa e a sua implementação prática, isto é, o programa propriamente dito e o seu texto, se o tiver. Se for o leitor a tratar disto, pagará apenas as taxas legais. Se preferir não ter o trabalho, recorra ao seu advogado ou a uma firma especializada de reconhecida idoneidade.

Seja programa em BASIC, seja em código de máquina, um método geralmente eficaz consiste em, tendo todo o programa em memória, gravar os 64K. Exemplos:

a) programa em BASIC que arranca na linha XXXX: prima, como comando directo, a seguinte linha:

```
SAVE "NOME" CODE 0,65536: GO TO XXXX
```

(o motivo de GO TO reside na hipótese de ter as variáveis na memória, mas não definidas no programa; se não for o caso RUN XXXX é suficiente).

b) programa em MC que arranca no endereço XXXX: prima, como comando directo, a seguinte linha:

```
SAVE "NOME" CODE 0,65536: RANDOMIZE XXXX
```

Se disse geralmente é porque, por vezes, o STACK fica baralhado com isto.

Outra maneira é fazer uns POKE's a algumas variáveis do sistema, como:

a) POKE 23659,0; elimina as linhas de input e reportagem de mensagens do sistema operativo. Não deve ser usado se for seguido adiante de instruções CLEAR ou CLS, ou se o programa solicitar inputs ou der reportagens nestas linhas, pois a consequência é o crash do sistema.

b) POKE 23613,0; deve ser incluído na primeira linha de BASIC e de preferência ser a primeira instrução. Consequência: tentativas de BREAK limpam toda a memória.

c) POKE 23757,0; POKE 23758,0; inútil para programas que utilizem a INTERFACE 1 e /ou os MICRODRIVES. De resto impede MERGE de programas em BASIC.

A última protecção que referirei refere-se à gravação de programas sem header. Vimos que tal é possível com MC no capítulo 21, quando discutimos algumas rotinas da ROM. Podemos preparar um programa inicial que carregue o verdadeiro programa de modo a que ele leia e despreze um falso header, mas leia e aceite o bloco seguinte, que é o verdadeiro programa, mas sem header algum.

Neste tema podemos fazer inúmeras variantes que, por motivos óbvios, deixarei ao cuidado do leitor.

Consciente de que falei e revelei mais protecções de software para o Spectrum do que qualquer outro autor, termino este capítulo com uma nota breve: há mais!

No capítulo seguinte vamos ver como se usam interrupts.

## ROTINA OFF

16K		48K	
	ORG# 7F4F		ORG # FE5D
7E4F 3E3E	LD A, # 3E	FE5D 3E3E	LD A, # 3E ; VECTOR EM A
7E51 ED56	IM 1	FE5F ED56	IM 1 ; MODO 1
7E53 ED47	LD I,A	FE61 ED47	LD I,A
7E55 C9	RET	FE63 C9	RET

## CAPÍTULO 24 USANDO O MODO 2 DE INTERRUPT

No capítulo 16 tivemos a oportunidade de ver os diferentes modos de interrupts e as instruções fundamentais aos mesmos. Agora veremos como construir os três programas necessários para o seu uso. Para tal, construiremos um programa com uma finalidade útil — permitir a leitura de BREAK num programa em MC — pois muitas vezes o erro comum é entrarmos num loop sem fim, do qual só se sai com a perda da informação em memória, desligando o Spectrum.

**PROBLEMA:** construir uma rotina que coloque o vector desejado no registo I, de modo a passar para essa rotina substituta o controle de uma função específica.

Vamos à tabela de vectores no apêndice, para escolher um apropriado para a versão de 16K e outro para a de 48K, de modo a que a rotina substituta fique tão alta na memória quanto possível. Uma rápida leitura permite-nos constatar que os vectores que nos interessam são, respectivamente, # 28 e # 09, iniciando a rotina substituta em #7E5C e # FE69, mais uma vez respectivamente. Iniciaremos a rotina "ON" 20 byte antes do endereço escolhido.

**PROBLEMA:** Construir uma rotina que nos permita voltar ao modo 1, (o habitual do Spectrum). Colocar a rotina a seguir à rotina "ON".

## ROTINA ON

16K		48K	
	ORG# 7E47		ORG #FE56
7E47 3E28	LD A, # 28	FE56 3E09	LD A, # 09 ; VECTOR EM A
7E49 ED47	LD I,A	FE58 ED47	LD I,A
7E4B ED5E	IM 2	FE5A ED5E	IM 2 ; MODO 2
7E4E C9	RET	FE5C C9	RET

**PROBLEMA:** Construir uma rotina substituta de interrupt que permita BREAK, sendo obtido premindo SYMBOL SHIFT e SPACE.

## ROTINA SUBSTITUTA

16K

48K

7E5C FF	ORG # 7E5C	FE69 FF	ORG # FE69
7E5D F3	RST # 38	FE6A F3	RST # 38
7E5E F5	DI	FE6B F5	DI
7E5F C5	PUSH AF	FE6C C5	PUSH AF
7E60 D5	PUSH BC	FE6D D5	PUSH BC
7E61 E5	PUSH DE	FE6E E5	PUSH DE
7E62 DDE5	PUSH HL	FE6F DDE5	PUSH HL
7E64 01FE7E	PUSH IX	FE71 01FE7E	PUSH IX
7E67 ED78	LD BC,# 7EFE	FE74 ED78	LD BC,# 7EFE
7E69 FEFC	IN A,(C)	FE76 FEFC	IN A,(C)
7E71 2808	CP # FC	FE78 2808	CP # FC
7E73 DDE1	JR Z,BREAK	FE7A DDE1	JR Z,BREAK
7E75 E1	POP IX	FE7C E1	POP IX
7E76 D1	POP HL	FE7D D1	POP HL
7E77 C1	POP DE	FE7E C1	POP DE
7E78 F1	POP BC	FE7F F1	POP BC
7E79 FB	POP AF	FE80 FB	POP AF
7E7A C9	EI	FE81 C9	EI
7E7B 01FE7E BREAK	RET	FE82 01FE7E BREAK	RET
7E7E ED78	LD BC,#7EFE	FE85 ED78	LD BC,# 7EFE
7E80 FEFC	IN A,(C)	FE87 FEFC	IN A,(C)
7E82 28F7	CP # FC	FE89 28F7	CP # FC
7E84 FB	JR Z,BREAK	FE8B FB	JR Z,BREAK
7E85 CF	EI	FE8C CF	EI
7E86 14	RST # 08	FE8D 14	RST # 08
	DEFB # 14		DEFB # 14

de usar este programa conjuntamente com outro que recorra a interrupts, terá como resultado um provável crash ou REST do sistema!

Reparou que a rotina testa duas vezes a combinação que permite o BREAK (que funciona mesmo com MC): a primeira passa o controle para a rotina de BREAK, caso a combinação tenha sido premida; a segunda aguarda que se retire os dedos da dita combinação antes de fazer o BREAK. Note também na leitura do teclado (RST # 38) antes do DI e os PUSH's e POP's para permitir reentrância. A leitura é confirmada com IN A,(C). Finalmente a directiva DEFB indica qual o erro ao sistema operativo.

Sempre que usar a tecla SYMBOL SHIFT é qualquer outra, ouvirá o som de memória cheia. Não se preocupe; isto deve-se à passagem pelo interrupt.

Sugestão: construa uma rotina substituta que lhe permita tirar cópias na impressora (se a tem) do estado do écran num momento qualquer.

No próximo capítulo vamos ver alguma coisa sobre o uso da calculadora de vírgula flutuante do Spectrum.

O uso da rotina é simples: ligamos ou desligamos a rotina substituta com RANDOMIZE USR NNNNN, conforme apropriado. A partir do momento em que o interrupt está ligado, premindo SYMBOL SHIFT e SPACE simultaneamente fará BREAK, com a mensagem de erro apropriado. Note, no entanto, que qualquer tentativa de usar este programa conjuntamente com outro que recorra a interrupts, terá como resultado um provável crash ou RESET do sistema!

## CAPÍTULO 25

### A CALCULADORA

Importa dizer desde já que este capítulo e o próximo se destinam ao leitor com alguma proficiência em MC. Aqueles que estão ainda nos seus primeiros passos terão, provavelmente, uma certa dificuldade em assimilar o que aqui se dirá. Para eles, estes capítulos finais são de interesse imediato limitado, pois a calculadora do Spectrum é constituída por rotinas algo complexas.

A calculadora tem 82 rotinas e compreende uma zona de memória e um stack próprios. Sem ela, o Spectrum seria uma máquina muito pobre.

O acesso à calculadora faz-se através da instrução RST # 28, sendo esta seguida de bytes definidos que indicam as rotinas que vão ser usadas. O último byte definido é SEMPRE # 38 (fim de cálculo).

O stack da calculadora é similar ao stack do Z80. O seu endereço base pode ser recolhido na variável do sistema STKBOT (de STack BOTtom) e a variável STKEND indica o endereço do primeiro byte livre acima do stack. Assim, se este estiver vazio ambos os endereços serão iguais. Tal como o stack do Z80, o stack da calculadora não ficará muito feliz se tentar tirar de lá valores quan-

do não tem nenhum ou deixar lá "restos" não usados. Podemos usar o stack para colocarmos números de 5 byte ou descritores de 5 byte de strings, para que a calculadora os manipule.

A zona de memória da calculadora tem 30 byte reservados na área das variáveis do sistema (MEMBOT). Estes 30 byte podem ser colocados noutro ponto qualquer da RAM do utilizador, fazendo os POKE's adequados à variável MEM (23656 e 23657). De modo análogo ao stack, esta zona permite conter valores de 5 byte, mas até um máximo de seis. Seguindo a nomenclatura do livro "THE COMPLETE SPECTRUM ROM DISASSEMBLY", que já teve a oportunidade de recomendar, chamaremos a cada conjunto de 5 byte "mem — 0, mem — 1, mem — 2, mem — 3, mem — 4 e mem — 5".

Para colocarmos valores no stack temos um grupo de rotinas, a saber:

a) Inteiros entre 0 e 255 — Coloque o valor no Acumulador e chame a rotina STACK — A, # 2D28.

b) Inteiros entre 0 e 65535 — Coloque o valor do par BC e chame a rotina STACK — BC, em # 2D2B.

c) Números de vírgula flutuante (formato 5 bytes, conforme descrito nas pág.s 169 — 170 do Manual do Spectrum) — Coloque o expoente no Acumulador e a mantissa de 4 byte nos registos B, C, D e E e chame a rotina # 2AB6, STK — STORE.

d) Descritores de strings — Coloque o início em DE e o comprimento em BC e chame a rotina STACK — STORE, em # 2AB6.

Como se vê, a rotina STACK — STORE da ROM é usada tanto para os descritores das strings como para números de vírgula flutuante. As rotinas para retirar valores do stack da calculadora são:

a) Inteiros entre 0 e 255 — Chame a rotina FP — TO — A, em # 2DD5, que comprimirá o valor do topo do stack no Acumulador. Se a compressão não for bem sucedida (o valor comprimido



não cabe no Acumulador), a Carry flag será igual a 1, ficando reset (igual a zero) se tudo correr bem.

b) Inteiros entre 0 e 65535 — A rotina a chamar é a FP — TO — BC em # 2DA2, que comprimirá o valor no par BC. A Carry flag comporta-se como na rotina FP — TO — A, indicando ou não o excesso.

c) Números de vírgula flutuante ou descritores de strings — Chamando STK — FETCH, em # 2BF1, poderá recolher do stack um valor de 5 byte, que será colocado nos registos A, B, C, D e E.

d) Impressão de um valor como número decimal — Chamando PRINT — FP, em # 2DE3, retirará o valor do topo do stack e imprimi-lo-á na posição PRINT adequada, na notação decimal.

O uso da área de memória da calculadora é feito recorrendo a literais (bytes definidos). Enunciemos brevemente as rotinas da calculadora.

## LITERAL MNEMÓNICA FUNÇÃO

# 00 JUMP — TRUE	Executa um salto condicional se o número endereçado por DE for verdadeiro.
# 01 EXCHANGE	Troca de posição os dois números do topo do stack.
# 02 DELETE	“Apaga” o número no topo do stack.
# 03 SUBTRACT	Muda o sinal do subtraendo e continua em ADDITION.
# 04 MULTIPLY	Executa a multiplicação dos dois primeiros números.
# 05 DIVISION	Executa a divisão de dois números de vírgula flutuante (nota: esta rotina tem um “bug”: o byte da ROM # 3200 devia conter # DA e não # E1.
# 06 TO — POWER	Eleva o primeiro número, $\times$ , à potência do segundo, y.
# 07 OR	Executa a operação binária $\times$ OR y, dando $\times$ se $y=0$ ou 1 se y for diferente de zero.

# 08 NO — & — NO	Executa a operação binária $\times$ AND y, dando $\times$ se y for diferente de zero e o valor zero se y for igual a zero.
# 09 NO — L — EQL	Operação de comparação $\times$ menor ou igual a y.
# 0A NO — GR — EQ	Operação de comparação $\times$ maior ou igual a y.
# 0B NOS — NEQL	Operação de comparação $\times$ diferente de y.
# 0C NO — GRTR	Operação de comparação $\times$ maior que y.
# 0D NO — LESS	Operação de comparação $\times$ menor que y.
# 0E NOS — EQL	Operação de comparação $\times$ igual a y.
# 0F ADDITION	Executa a adição de dois números de vírgula flutuante.
# 10 STR — & — NO	Executa a operação binária $\times$ \$ AND y, dando $\times$ \$ se y for diferente de zero ou string nula no caso contrário.
# 11 STR — L — EQL	Estas rotinas, prefixadas pelos literais.
# 12 STR — GR — EQ	entre # 11 e # 16, são análogas às prefixadas
# 13 STRS — NEQL	pelos literais de # 09 a # 0E, mas onde os
# 14 STR — GRTR	termos de comparação são strings (ou melhor,
# 15 STR — LESS	descritores de strings) e não números
# 16 STRS — EQL	propriamente ditos.
# 17 STRS — ADD	Executa a operação binária $\times$ \$ + y\$.
# 18 VAL\$	Executa a função VAL\$ $\times$ \$ do BASIC.
# 19 USR — \$	Executa a função USR $\times$ \$ do BASIC.
# 1A READ — IN	Esta rotina permite a leitura de dados através de streams diferentes

# 1B NEGATE	dos standard no Spectrum. Troca o sinal do número ("unary minus").	# 34 STK — DATA	Esta subrotina coloca no topo do stack um número de vírgula flutuante que lhe seja fornecido como 2, 3, 4, ou 5 literais.
# 1C CODE	Executa a função CODE× \$ do BASIC.	# 35 DEC — JR — NZ	Efectua uma operação DJNZ, mas o contador não é o registo B, antes a variável do sistema operativo BREG.
# 1D VAL	Executa a função VAL× do BASIC.	# 36 LESS — 0	Retorna no topo do stack o valor 1 se o último valor for menor que zero ou zero nos outros casos.
# 1E LEN	Executa a função LEN× \$ do BASIC.	# 37 GREATER — 0	Retorna no topo do stack o valor 1 se o último valor for maior que zero ou zero nos casos restantes.
# 1F SIN	Executa a função SIN× do BASIC.	# 38 END — CALC	Termina uma chamada a RST # 28
# 20 COS	Executa a função COS× do BASIC.	# 39 GET — ARGV	Esta subrotina transforma o argumento× de SIN× ou COS× num valor v.
# 21 TAN	Executa a função TAN× do BASIC.	# 3A TRUNCATE	Retorna no topo do stack o resultado de uma truncagem inteira de× para zero.
# 22 ASN	Executa a função ASN× do BASIC.	# 3B FP — CALC — 2	Esta subrotina é usada pela rotina em # 2757 para fazer uma operação aritmética única.
# 23 ACS	Executa a função ACS× do BASIC.	# 3C E — TO — FP	Esta rotina converte um número na forma×Em, sendo m um inteiro positivo ou negativo, num número de vírgula flutuante.× deve estar no topo do stack e m no Acumulador.
# 24 ATN	Executa a função ATN× do BASIC.	# 3D RE — STACK	O número do topo do stack é passado à forma de vírgula flutuante.
# 25 LN	Executa a função LN× do BASIC.	# 3I SERIES	Seguida do literal # 86, # 88 ou # 8C gera a série de polinómios de Chebyshev, para aproximar SIN, ATN, LN e EXP, e derivar as funções que delas dependem.
# 26 EXP	Executa a função EXP× do BASIC.	# 3F STK	Seguida dum literal, de # A0 a # A5, coloca a constante, respecti-
# 27 INT	Executa a função INT× do BASIC.		
# 28 SQR	Executa a função SQR× do BASIC.		
# 29 SGN	Executa a função SGN× do BASIC.		
# 2A ABS	Executa a função ABS× do BASIC.		
# 2B PEEK	Executa a função PEEK× do BASIC.		
# 2C IN	Executa a função IN× do BASIC (usa IN,A(C))		
# 2D USR — NO	Executa a função USR× do BASIC.		
# 2E STR\$	Executa a função STR\$× do BASIC.		
# 2F CHR\$	Executa a função CHR\$× do BASIC.		
# 30 NOT	Dá resultado 1 se o valor do topo do stack for maior que zero e zero nos restantes casos		
# 31 DUPLICATE	Duplica o primeiro número do stack.		
# 32 N — MOD — M	Sendo m um inteiro positivo no topo do stack e n o segundo valor do stack, também inteiro, esta rotina coloca no topo do stack o quociente inteiro INT (n /m) e no segundo lugar o resto n — INT (n /m).		
# 33 JUMP	Executa um salto incondicional.		

vamente 0, 1, 0.5,  $\pi / 2$  e 10, no topo do stack.

# 40 ST — MEM      Seguida dum literal, de # C0 a # C5, copia o número do topo do stack para mem-n, onde n vai de 0 a 5. O stack não é afectado.

# 41 GET — MEM      Seguida dum literal, de # E0 a # E5, copia o número contido em mem-n para o topo do stack.

Depois deste breve resumo, vejamos um exemplo simples de uso directo da calculadora.

**PROBLEMA:** considere que os registos A, B, C, D e E contêm um número no formato de virgula flutuante. Calcule o valor y, resultante da função  $y = \sin x / \sin (\cos x)$ . Imprima y no écran em formato decimal.

Limitar-me-ei a dar a listagem em Z80 Assembler, uma vez que o leitor beneficiará do exemplo como ilustração do uso da calculadora. Note que a sua maior vantagem (no meu entender) é processar funções complicadas.

Para resolvermos o problema, a calculadora deverá calcular:

- $\sin x$
- $\cos x$
- $\sin (\cos x)$
- $\sin x / \sin (\cos x)$
- imprimir o resultado em decimal.

#### PROGRAMA

```
FUNÇÃO CALL #2AB6 ; o número é colocado no stack
RST # 28          ; entrada na calculadora
DEFB # 31         ; duplica
DEFB # 1F         ; SIN x
DEFB # 01         ; troca: SIN x é o segundo valor
DEFB # 20         ; COS x
DEFB # 1F         ; SIN (COS x)
DEFB # 05         ; divisão SIN x / SIN (COS x)
```

```
DEFB # 38         ; fim de cálculo
PRINT CALL #2DE3 ; imprime número no écran
RET
```

Como se pode ver, o seu uso é simples, desde que se tenha cuidado com a ordenação dos números no stack, isto é, desde que realize as operações de trás para a frente, como no exemplo.

O último capítulo tratará dos algoritmos de divisão e multiplicação.

## CAPÍTULO 26

### MULTIPLICANDO E DIVIDINDO

Como é do conhecimento do leitor, qualquer operação é redutível à adição, dentro do quadro aritmético. Como tal, a multiplicação e a divisão nada mais são do que somas (ou subtracções) sucessivas, isto é uma série de iterações ou loops.

O algoritmo que veremos prende-se com a resolução binária dos problemas. No caso da multiplicação, temos o problema de multiplicar por zero ou por um. Salta à vista que no primeiro caso obtemos zero e no segundo obtemos o multiplicando; isto traduz-se no algoritmo:

**SE O PRESENTE BIT DO MULTIPLICADOR É 1, SOMA-SE O MULTIPLICANDO AO PRODUTO PARCIAL.**

Se queremos que as coisas corram bem, teremos de considerar o alinhamento correcto dos números (tal como quando fazemos a multiplicação à mão). Para isso, bastam duas operações:

- a) Fazer um SHIFT à esquerda de um bit ao multiplicador para que o bit a examinar fique na Carry flag;
- b) Fazer um SHIFT à esquerda de um bit ao produto para que a adição seguinte seja correctamente alinhada.

**PROBLEMA:** multiplique um número de 8 - bit (sem sinal), colocado em # 9C40, pelo número de 8 - bit (sem sinal), colocado em # 9C41. Coloque o resultado em # 9C42 - # 9C43 (LSB primeiro). O programa deve ser relocável.

```
LD HL,# 9C40 ; inicializar HL
LD E, (HL)   ; transferir multiplicando para E
LD D,# 00    ; expandindo para 16-bit
INC HL
LD A,(HL)    ; multiplicador em A
INC HL
PUSH HL      ; guardar no stack o destino do resultado
LD HL,# 00   ; produto= zero
LD B,# 08    ; contador
MULTIP ADD HL,HL ; alinea b em efeito
      RLA      ; alinea a em efeito
      JR NC, LOOP ; a carry do multiplicador é 1?
      ADD HL,DE ; sim. Soma multiplicando ao produto
LOOP  DJNZ, MULTIP
      EX DE,HL ; produto em DE
      POP HL  ; destino do resultado
      LD (HL),E ; guarda LSB
      INC HL
      LD (HL),D ; guarda MSB
      RET
```

Conforme expressei, o programa é relocável. Mudando apenas a inicialização de HL, podemos recolher os números e colocar o resultado noutro ponto qualquer da memória.

A divisão obtém-se pela subtracção sucessiva, até ao ponto em que o que resta do dividendo é inferior ao divisor, obtendo-se assim o resto. Uma vez que estamos a proceder a operações binárias, o problema assenta em sabermos se o bit do quociente será zero ou um (se o que resta do dividendo pode ser subtraído do divisor ou não). Donde o algoritmo:

SE O DIVISOR PUDE SER SUBTRAÍDO DOS OITO BIT MAIS SIGNIFICATIVOS DO DIVIDENDO SEM “BORROW” (CARRY = 1), O BIT CORRESPONDENTE DO QUOCIENTE SERÁ 1; DE OUTRO MODO, SERÁ 0.

Analogamente à multiplicação, há que alinhar o dividendo e o quociente, fazendo SHIFT à esquerda a ambas antes de cada tentativa de subtração. Uma vez que 1 bit do dividendo é eliminado ao mesmo tempo que é determinado 1 bit do quociente, ambos podem partilhar um registo de 16 - bit.

PROBLEMA: divida um número de 16-bit (sem sinal), colocado em # 9C40 (LSB) e # 9C41 (MSB) pelo número de 8-bit (sem sinal), colocado em # 9C42. Coloque o quociente em # 9C43 e o resto em # 9C44. O programa deve ser relocatável.

```
LD HL,# 9C40 ; inicializar HL
LD E, (HL) ; transferir LSB do dividendo para E
INC HL
LD D, (HL) ; transferir MSB do dividendo para D
INC HL
LD A,(HL) ; transferir divisor para A
INC HL
EX DE, HL ; guardar em DE o destino do quociente e
; em HL o dividendo
LD C, A ; divisor em C
LD B,# 08 ; contador
DIVISA ADD HL,HL ; shift à esquerda em efeito
LD A,H
SUB C ; o divisor pode ser subtraído?
JR C,LOOP ; não. Segue para LOOP
LD H,A ; sim. Subtraia divisor do dividendo e
INC L ; some 1 ao quociente
LOOP DJNZ,DIVISA
EX DE, HL ; quociente e resto em DE
LD (HL), D ; guarda quociente na memória
INC HL
LD (HL), E ; guarda o resto na memória
RET
```

É evidente que, para divisões mais longas usaríamos a instrução SBC HL. Note que este programa obedece, nas suas linhas gerais, ao mesmo tipo de raciocínio que especifiquei no início: todas as operações aritméticas são redutíveis à adição.

Terminamos aqui esta “visita guiada” ao Z80 Assembler, com particular incidência no ZX SPECTRUM, como suporte de hardware. Não fique por aqui! Consulte a bibliografia, onde poderá encontrar obras mais avançadas do que esta e que contribuirão para o seu aperfeiçoamento no domínio do CPU de 8 - bit mais complicado do mercado de hoje.

TABELA DE CONVERSÃO DECIMAL — HEXADECIMAL  
LSB: DECIMAL 0 — 255 HEXADECIMAL 00 — FF

DEC.	HEX.	DEC.	HEX.	DEC.	HEX.	2'sC.	DEC.	HEX.	2'sC.
0	00	64	40	128	80	- 128	192	C0	- 64
1	01	65	41	129	81	- 127	193	C1	- 63
2	02	66	42	130	82	- 126	194	C2	- 62
3	03	67	43	131	83	- 125	195	C3	- 61
4	04	68	44	132	84	- 124	196	C4	- 60
5	05	69	45	133	85	- 123	197	C5	- 59
6	06	70	46	134	86	- 122	198	C6	- 58
7	07	71	47	135	87	- 121	199	C7	- 57
8	08	72	48	136	88	- 120	200	C8	- 56
9	09	73	49	137	89	- 119	201	C9	- 55
10	0A	74	4A	138	8A	- 118	202	CA	- 54
11	0B	75	4B	139	8B	- 117	203	CB	- 53
12	0C	76	4C	140	8C	- 116	204	CC	- 52
13	0D	77	4D	141	8D	- 115	205	CD	- 51
14	0E	78	4E	142	8E	- 114	206	CE	- 50
15	0F	79	4F	143	8F	- 113	207	CF	- 49
16	10	80	50	144	90	- 112	208	D0	- 48
17	11	81	51	145	91	- 111	209	D1	- 47
18	12	82	52	146	92	- 110	210	D2	- 46
19	13	83	53	147	93	- 109	211	D3	- 45
20	14	84	54	148	94	- 108	212	D4	- 44
21	15	85	55	149	95	- 107	213	D5	- 43
22	16	86	56	150	96	- 106	214	D6	- 42
23	17	87	57	151	97	- 105	215	D7	- 41
24	18	88	58	152	98	- 104	216	D8	- 40
25	19	89	59	153	99	- 103	217	D9	- 39
26	1A	90	5A	154	9A	- 102	218	DA	- 38
27	1B	91	5B	155	9B	- 101	219	DB	- 37
28	1C	92	5C	156	9C	- 100	220	DC	- 36
29	1D	93	5D	157	9D	- 99	221	DD	- 35
30	1E	94	5E	158	9E	- 98	222	DE	- 34
31	1F	95	5F	159	9F	- 97	223	DF	- 33

BIBLIOGRAFIA

LEVENTHAL, LANCE A. — Z80 ASSEMBLY LANGUAGE  
PROGRAMMING  
OSBORNE /MCGRAW — HILL,  
U.S.A., 1979

LEVENTHAL, LANCE A, & SAVILLE, WINTHROP — Z80 AS-  
SEMBLY LANGUAGE SUBROUTINES OS-  
BORNE /MCGRAW — HILL, U.S.A., 1983

LOGAN, DR. IAN & O'HARA, DR. FRANK — THE COMPLE-  
TE SPECTRUM ROM DISASSEMBLY  
MELBOURNE HOUSE PUBLISHERS UNI-  
TED KINGDOM, 1983

32	20	96	60	160	A0	- 96	224	E0	- 32
33	21	97	61	161	A1	- 95	225	E1	- 31
34	22	98	62	162	A2	- 94	226	E2	- 30
35	23	99	63	163	A3	- 93	227	E3	- 29
36	24	100	64	164	A4	- 92	228	E4	- 28
37	25	101	65	165	A5	- 91	229	E5	- 27
38	26	102	66	166	A6	- 90	230	E6	- 26
39	27	103	67	167	A7	- 89	231	E7	- 25
40	28	104	68	168	A8	- 88	232	E8	- 24
41	29	105	69	169	A9	- 87	233	E9	- 23
42	2A	106	6A	170	AA	- 86	234	EA	- 22
43	2B	107	6B	171	AB	- 85	235	EB	- 21
44	2C	108	6C	171	AC	- 84	236	EC	- 20
45	2D	109	6D	173	AD	- 83	237	ED	- 19
46	2E	110	6E	174	AE	- 82	238	EE	- 18
47	2F	111	6F	175	AF	- 81	239	EF	- 17
48	30	112	70	176	B0	- 80	240	F0	- 16
49	31	113	71	177	B1	- 79	241	F1	- 15
50	32	114	72	178	B2	- 78	242	F2	- 14
51	33	115	73	179	B3	- 77	243	F3	- 13
52	34	116	74	180	B4	- 76	244	F4	- 12
53	35	117	75	181	B5	- 75	245	F5	- 11
54	36	118	76	182	B6	- 74	246	F6	- 10
55	37	119	77	183	B7	- 73	247	F7	- 9
56	38	120	78	184	B8	- 72	248	F8	- 8
57	39	121	79	185	B9	- 71	249	F9	- 7
58	3A	122	7A	186	BA	- 70	250	FA	- 6
59	3B	123	7B	187	BB	- 69	251	FB	- 5
60	3C	124	7C	188	BC	- 68	252	FC	- 4
61	3D	125	7D	189	BD	- 67	253	FD	- 3
62	3E	126	7E	190	BE	- 66	254	FE	- 2
63	3F	127	7F	191	BF	- 65	255	FF	- 1

TABELA DE CONVERSÃO DECIMAL — HEXADECIMAL  
MSB: DECIMAL 0 — 65280 HEXADECIMAL 00 — FF

DEC.	HEX.	DEC.	HEX.	DEC.	HEX.	DEC.	HEX.
0	00	16384	40	32768	80	49152	C0
256	01	16640	41	33024	81	49408	C1
512	02	16896	42	33280	82	49664	C2
768	03	17152	43	33536	83	49920	C3
1024	04	17408	44	33792	84	50176	C4
1280	05	17664	45	34048	85	50432	C5
1536	06	17920	46	34304	86	50688	C6
1792	07	18176	47	34560	87	50944	C7
2048	08	18432	48	34816	88	51200	C8
2304	09	18688	49	35072	89	51456	C9
2560	0A	18944	4A	35328	8A	51712	CA
2816	0B	19200	4B	35584	8B	51968	CB
3072	0C	19456	4C	35840	8C	52224	CC
3328	0D	19712	4D	36096	8D	52480	CD
3584	0E	19968	4E	36352	8E	52736	CE
3840	0F	20224	4F	36608	8F	52992	CF
4096	10	20480	50	36864	90	53248	D0
4352	11	20736	51	37120	91	53504	D1
4608	12	20992	52	37376	92	53760	D2
4864	13	21248	53	37632	93	54016	D3
5120	14	21504	54	37888	94	54272	D4
5376	15	21760	55	38144	95	54528	D5
5632	16	22016	56	38400	96	54784	D6
5888	17	22272	57	38656	97	55040	D7
6144	18	22528	58	38912	98	55296	D8
6400	19	22784	59	39168	99	55552	D9
6656	1A	23040	5A	39424	9A	55808	DA
6912	1B	23296	5B	39680	9B	56064	DB
7168	1C	23552	5C	39936	9C	56320	DC
7424	1D	23808	5D	40192	9D	56576	DD
7680	1E	24064	5E	40448	9E	56832	DE
7936	1F	24320	5F	40704	9F	57088	DF

8192	20	24576	60	40960	A0	57344	E0
8448	21	24832	61	41216	A1	57600	E1
8704	22	25088	62	41472	A2	57856	E2
8960	23	25344	63	41728	A3	58112	E3
9216	24	25600	64	41984	A4	58368	E4
9472	25	25856	65	42240	A5	58624	E5
9728	26	26112	66	42496	A6	58880	E6
9984	27	26368	67	42752	A7	59136	E7
10240	28	26624	68	43008	A8	59392	E8
10496	29	26880	69	43264	A9	59648	E9
10752	2A	27136	6A	43520	AA	59904	EA
11008	2B	27392	6B	43776	AB	60160	EB
11264	2C	27648	6C	44032	AC	60416	EC
11520	2D	27904	6D	44288	AD	60672	ED
11776	2E	28160	6E	44544	AE	60928	EE
12032	2F	28416	6F	44800	AF	61184	EF
12288	30	28672	70	45056	B0	61440	F0
12544	31	28928	71	45312	B1	61696	F1
12800	32	29184	72	45568	B2	61952	F2
13056	33	29440	73	45824	B3	62208	F3
13312	34	29696	74	46080	B4	62464	F4
13568	35	29952	75	46336	B5	62720	F5
13824	36	30208	76	46592	B6	62976	F6
14080	37	30464	77	46848	B7	63232	F7
14336	38	30720	78	47104	B8	63488	F8
14592	39	30976	79	47360	B9	63744	F9
14848	3A	31232	7A	47616	BA	64000	FA
15104	3B	31488	7B	47872	BB	64256	FB
15360	3C	31744	7C	48128	BC	64512	FC
15616	3D	32000	7D	48384	BD	64768	FD
15872	3E	32256	7E	48640	BE	65024	FE
16128	3F	32512	7F	48896	BF	65280	FF

## TABELA DE ESCOLHA DE ENDEREÇO PARA ROTINA DE INTERRUPT

(MODO 2)

FÓRMULA:  $I * 265 + 255$  — O valor obtido será o endereço do byte que, conjuntamente com o imediatamente a seguir, darão o endereço da rotina substituta.

NOTA: este valor terá de ser recolhido na ROM para não criar “neve” no écran.

EXEMPLO:  $I = 9$  — (End. 2559 decimal (na ROM) byte 2559 = #69; byte 2560 = #FE ou seja, a rotina substituta começa em #FE69 ou 65129 decimal.

VALORES PARA I (VECTORES)		INÍCIO DA ROTINA SUBSTITUTA	
DEC.	HEX.	DEC.	HEX.
1	01	52818	CE52
2	02	22269	56FD
3	03	39020	986C
6	06	29149	71DD
9	09	65129	FE69
10	0A	32802	8022
11	0B	58888	E608
12	0C	53183	CFBF
14	0E	52503	CD17
15	0F	27928	6D18
16	10	51984	CB10
18	12	52481	CD01



19	13	49749	C255
20	14	25705	6469
21	15	51673	C9D9
22	16	51568	C970
25	19	23842	5D22
28	1C	49947	C31B
30	1E	26573	67CD
32	20	52513	CD21
33	21	33485	82CD
35	23	49537	C181
40	28	32348	7E5C
41	29	58154	E32A
48	30	60208	EB30
49	31	57640	E128
53	35	57124	DF24
54	36	34307	8603
55	37	41231	A10F

## GLOSSÁRIO DE TERMOS E ABREVIATURAS

**ASSEMBLER** — linguagem de nível 1. Corresponde a Código de Máquina na razão de 1 para 1.

**BIT** — unidade mínima de informação. Deriva de BInary digiT. Um bit corresponde a um sinal ou a uma ausência de sinal, no caso a uma voltagem ou ausência de voltagem. Representa-se o seu estado pela notação binária: 0 para ausência de sinal (bit reset), 1 para presença de sinal (bit set).

**BYTE** — conjunto de oito bit. Um byte permite designar qualquer número entre 0 e 255, isto é, permite representar 256 códigos de informação distintos. Note que dois byte podem representar, como número máximo, 65535.

**CÓDIGO DE MÁQUINA** — abreviado MC ou M/C. Linguagem de nível 0. É a linguagem do microprocessador e nada tem de humano.

**ENDEREÇO** — por endereço entende-se um número de 2 byte que designa um local de memória. Ex: o endereço 23296 marca o início do buffer da impressora.

**FONTE** — também designado programa-fonte (do inglês source). Programa numa linguagem de nível 1 ou superior.

**LSB** — do inglês Less Significant Byte, ou seja, o byte menos significativo. Também designado por LOB. Considerando o endereço #9C40, #40 é o LSB.

**MSB** — do inglês Most Significant Byte, ou seja, o byte mais significativo. Também designado por HOB. Considerando o endereço #9C40, #9C é o MSB.

**NIBBLE** — meio byte (4 bit).

**NOTAÇÃO BINÁRIA** — sistema numérico de base 2, cujos algarismos são 0 e 1. Ex: o número 5 decimal representa-se por 101 em binário. Um número nesta notação é precedido por % ou seguido de B. Ex: %101 = 101B = 5 decimal.

**NOTAÇÃO DECIMAL** — sistema numérico de base 10, usado pelo comum dos mortais. Os números nesta notação ou são seguidos de DEC. ou (a maior parte das vezes) apresentados sem sufixos ou prefixos. Ex: 35000.

## ABREVIATURAS

dado ou data	— um número de 8-bit.
dado 16 ou data 16	— um número de 16-bit.
d	— deslocamento (offset).
end	— endereço de memória.
flag: C	— Carry
Z	— Zero
S	— Sign (sinal)
P / O	— Parity / Overflow (paridade / excesso)
AC	— Carry Auxiliar
N	— Subtract
flag: estado: NA	— não afectada
0	— toma o valor 0 (reset)
1	— toma o valor 1 (set)
X	— modificada de modo a reflectir o resultado da operação
?	— efeito desconhecido
indreg	— registo de indexamento (IY ou IX).
par	— par de registos (BC, DE, HL ou AF).
pareg	— par de registos (BC, DE, HL ou SP).
(NN)	— valor do conteúdo do byte contido entre parêntesis, podendo ser de 8 ou 16 bit, conforme o operador.

## NOTAÇÃO

**HEXADECIMAL** — sistema numérico de base 16; usa os algarismos de 0 a 9 e as letras de A a F. Os números nesta notação são precedidos por # ou seguidos de H ou HEX. Ex: #5B00 = 5B00H = 5B00 HEX.

**OCTETO** — conjunto de 8 byte.

**RAM** — do inglês Random Access Memory (memória de acesso aleatório). Designa o tipo de memória que pode ser lida, apagada e escrita. O seu conteúdo perde-se se o computador for desligado. Em termos práticos é a memória que o utilizador tem disponível para uso.

**ROM** — do inglês Read Only Memory (memória de leitura). Designa a memória que pode ser lida, mas não apagada ou re-escrita. A falta de energia não a afecta. Contém o sistema operativo do Spectrum.

**Z80** — nome do microprocessador usado pelo Spectrum. É um processador de 8-bit, isto é, a unidade de palavra é de 8-bit de comprimento.

## ÍNDICE

INTRODUÇÃO .....	9
CAPÍTULO 1. PROBLEMAS A ENFRENTAR .....	11
Programa Muda Base .....	14
CAPÍTULO 2. LINGUAGENS DE ALTO E BAIXO NÍVEL ...	17
CAPÍTULO 3. ASSEMBLADORES .....	21
CAPÍTULO 4. O Z80/REGISTOS E «FLAGS» .....	28
CAPÍTULO 5. LOAD .....	33
CAPÍTULO 6. OPERAÇÕES ARITMÉTICAS .....	39
Operações Aritméticas de 8 BIT .....	39
Operações Aritméticas de 16 BIT .....	44
CAPÍTULO 7. INSTRUÇÕES LÓGICAS .....	48
Grupo AND .....	48
Grupo OR .....	49
Grupo XOR .....	50
CAPÍTULO 8. INSTRUÇÕES DE SALTO .....	53
Grupo JR .....	53
Grupo JP .....	54
Grupo CALL .....	55
CAPÍTULO 9. LOOPS .....	57
CAPÍTULO 10. TRANSFERÊNCIA E PESQUISA DE BLOCOS	61
Grupo LD .....	61
Grupo CP .....	63
CAPÍTULO 11. OPERAÇÕES DE STACK .....	66
CAPÍTULO 12. TROCAS .....	69
CAPÍTULO 13. BITS .....	71
CAPÍTULO 14. ROTAÇÕES E MUDANÇAS .....	74
CAPÍTULO 15. O MUNDO EXTERIOR-IN E OUT .....	81
Grupo IN .....	81
Grupo OUT .....	83
CAPÍTULO 16. INTERRUPÇÕES .....	85
Modo 0 .....	85
Modo 1 .....	86
Modo 2 .....	86
CAPÍTULO 17. NOTAÇÃO BCD .....	89

CAPÍTULO 18. O RESTO-INSTRUÇÕES DE STATUS .....	92
Instruções sem Designação Especial .....	92
Complementando o Acumulador .....	93
CAPÍTULO 19. HEX LOADER .....	94
CAPÍTULO 20. APRENDENDO A SOMAR .....	98
CAPÍTULO 21. ALGUMAS ROTINAS ÚTEIS DA ROM .....	105
CAPÍTULO 22. MANIPULANDO O ÉCRAN .....	113
O DIF .....	114
LOAD "SCREEN\$ .....	114
O ATF .....	117
CAPÍTULO 23. PROTECÇÃO DE SOFTWARE .....	119
CAPÍTULO 24. USANDO O MODO 2 DE INTERRUPT .....	112
Rotina ON .....	123
Rotina OF .....	123
Rotina Substituta .....	124
CAPÍTULO 25. A CALCULADORA .....	126
Programa .....	132
CAPÍTULO 26. MULTIPLICANDO E DIVIDINDO .....	134
BIBLIOGRAFIA .....	138
TABELA DE CONVERSÃO DECIMAL .....	139
TABELA DE ESCOLHA DE ENDEREÇO PARA ROTINA DE INTERRUPT .....	143
GLOSSÁRIO DE TERMOS E ABREVIATURAS .....	145
Abreviaturas .....	147

Este livro acabou de se imprimir  
 em 1985  
 para a  
 EDITORIAL PRESENÇA, LDA.  
 na  
 Empresa Gráfica Feirense, Lda.  
 Vila da Feira